TEKNILLINEN KORKEAKOULU
TEKNISKA HÖGSKOLAN
HELSINKI UNIVERSITY OF TECHNOLOGY

Jorge Latorre de la Calle

Robótica Asistencial: Soporte Robótico de Apoyo

Proyecto Fin de Carrera en Ingeniería Industrial
Espoo, Diciembre de 2009
Supervisor: Panu Harmo MSc.

Resumen del Proyecto Fin de Carrera

Autor: Jorge Latorre de la Calle
Título original: Assistive Robotics and Getting Up: Support pole

Helsinki University of Technology

Departamento: Automation and Systems Technology.
Profesorado: AS-84 Automation Technology.

Fecha: Diciembre de 2009 **Lugar:** Espoo

Este trabajo forma parte del proyecto “Ylämummo Heilahtaa” Mide Technology Project for Students

Supervisor: Master of Science Panu Harmo

Este proyecto trabaja sobre el prototipo de soporte robótico desarrollado originalmente por el estudiante Teemu Kuusisto en su Master's Thesis. El prototipo consiste en una barra con una agarradera que se mueve sobre un raíl situado en el techo de su estructura de pruebas. Este soporte puede extenderse y contraerse entre el suelo y el techo. Cuando está totalmente extendido/tensado, puede usarse como soporte para ayudar a ponerse en pie de sus usuarios desde sillas, camas, etc. Un mando de control permite al usuario determinar el movimiento y tensión del soporte.

Se realizan mejoras sobre el código original, corrigiendo errores de software y añadiendo funcionalidades adicionales, como detección de colisiones y control asistido básico. Finalmente, se añade al sistema un modulo de reconocimiento de voz, permitiendo al usuario manejar el soporte robótico sin necesitar el mando de control.

Palabras clave: Dispositivos asistivos, Robótica Asistencial Social, cuidado de personal de avanzada edad y/o con discapacidad física, reconocimiento de voz, automatización, ponerse en pie.

Antes de empezar con el proyecto en sí mismo, me gustaría dar las gracias a todos aquellos que me han ayudad, guiado o simplemente hecho reír en los momentos más duros. A todos aquellos que, en definitiva, me han permitido escribir este mismo proyecto. Sin ellos, no hubiera sido lo mismo.

Gracias a:

Panu Harmo, por introducirme en el campo de la robótica asistida, permitirme trabajar en este proyecto y guiarme durante todo el proceso y, por supuesto, por hacer más cálida mi estancia en la universidad.

Anja Hänninen, por ayudarme con todo el papeleo y burocracia que una beca de este estilo conlleva, por aguantarme y darme consejos para poder quedarme en Finlandia. Me sentí realmente un miembro más del departamento, no sólo un visitante o estudiante de intercambio.

Johannes Aalto, por ayudarme con todos los problemas técnicos del proyecto, especialmente durante las primeras semanas, y también en algunos momentos de falta de lucidez en las últimas.

José Vallet, por darme fuerza para perseguir mis objetivos y ayudarme a entender el estilo de vida Finés.

A todos aquellos amigos que han estado ahí, al otro lado del pasillo, a un viaje de autobús o en el otro extreme de la línea por teléfono/Skype, dándome calor en los peores momentos y compartiendo mi felicidad en los mejores

Helsinki University of Technology, el Departamento de Sistemas y Automática y la Universidad Carlos III de Madrid, por permitirme vivir la experiencia Erasmus: Descubrir un país y una cultura totalmente nuevos para mí, finalizar mis estudios en el extranjero en una gran universidad y conocer a tanta gente inolvidable en el proceso.

Finlandia y los Fineses, por recibirme con los brazos abiertos y ser un país y una gente tan maravillosa... Es una pena que no haya podido estar aquí por más tiempo... Pero estoy seguro de que volveré algún día de una u otra forma... Lo prometo...

Índice

Términos y Acrónimos

A/D: Analógico a Digital. Referente a la conversión de señales analógicas en señales digitales.

AC: Alternate Current. Corriente eléctrica que invierte periódicamente la dirección de su flujo.

AR: Assistive Robotics. Robots que asisten a las personas con discapacidades físicas a través de interacción física.

AVR: AVR es una arquitectura tipo Harvard de 8-bit RIDC de microcontroladores en un sólo chip que fue desarrollada por Atmel en 1996. AVR fue una de las primeras familias de microcontroladores con la capacidad de usar memoria flash para el almacenamiento de programas en un sólo chip.

C/C++: Ampliamente usado, se trata de un lenguaje de programación procedural de uso general usado tanto en sistemas operativos como en aplicaciones. C++ es simplemente una evolución de C.

CAN: Controller Area Network. Interfaz periférico diseñado para su uso en aplicaciones de automoción industrial.

CANbus: Protocolo del bus de comunicación del Controller Area Network (CAN).

CPU: Central Processing Unit. Parte central de una computadora que realice las principales funciones de procesamiento.

DB9: Tipo de conector usado de manera habitual para la comunicación en serie de ordenadores utilizando RS-232.

DC: Direct current. Corriente eléctrica fluyendo en una única dirección.

GUI: Graphical User Interface. Interfaz entre el usuario y el sistema informático o aplicación basado en la presentación de la información de manera gráfica en lugar de texto.

GPL: General Public License. Tipo de licencia de software libre de uso más habitual.

I/O: Input/Output. El acto de mover información a o desde un computador o módulo.

I2C: Inter Integrated-Circuit. Bus multi-maestro de tipo serie usado para la adición de periféricos simples y de baja velocidad a una placa base, sistema integrado o teléfono móvil.

IDE: Integrated Development Environment o Entorno de Desarrollo Integrado. Se trata de una aplicación de software que proporciona una gran cantidad de ventajas y facilidades a los programadores en su trabajo de desarrollo de software.

OS: Operative System. Sistema Operativo.

PC: Personal Computer. Computadora diseñada para ser usada por un único usuario a la vez.

PCB: Printed Circuit Board. Placa diseñada para el soporte, sujeción y conexión eléctrica de los componentes electrónicos de un modulo o sistema.

ROM: Read-Only Memory. Memoria no volátil de un sistema electrónico en la cual los programas y sistema operativo son almacenados.

RS-232: Interfaz estándar para la comunicación en serie de dispositivos electrónicos.

SAR: Socially Assistive Robotics. Intersección entre la Robótica Asistida (AR) y la Robótica Social Interactiva (SIR).

SIR: Socially Interactive Robotics. Robots cuyo propósito es el desarrollo de alguna forma de interacción humano-robot.

UART: Universal Asynchronous Receiver/Transmitter. Dispositivo en una computadora o modulo que transforma información en serie a paralelo y viceversa.

Listado de figuras

1. Introducción

1.1.- Nuestro Futuro[1]

Antes de adentrarnos en explicaciones complejas sobre el proyecto, es necesario echar un vistazo a las estimaciones que se tienen sobre el futuro de la población mundial. La siguiente tabla muestra la población mayor de 65 años, edad que coincide con la de jubilación en muchos países, en relación al resto, región por región. (Figura (1.1)):

	Year		
Region	**2000**	**2015**	**2030**
Asia	6	7.8	12
North Africa/Near East	4.3	5.3	8.1
Sub-Saharan Africa	2.9	3.2	3.7
Europe	15.5	18.7	24.3
North America	12.6	14.9	20.3
Latin America/Caribbean	5.5	7.5	11.6
Oceania	10.2	12.4	16.3

(1.1) Población mayor de 65 años (Instituto de estadística de EEUU, 2000)

Tal y como se muestra en la figura, la población mundial de una edad superior a 65 años se va a duplicar en tan sólo 30 años. La población de mayor edad, superior a 85 años, es el segmento que más rápidamente está creciendo, tanto en Europa como en Norteamérica. Aunque esto es en realidad una buena señal, indicativo directo de la mejora del nivel de vida, es también la causa de un problema social que ha de ser soliviantado. Así pues, hay una fuerte correlación entre la discapacidad física y la edad: Se estima que al menos el 62% de los ancianos mayor de 85 años tienen grandes dificultades a la hora de realizar una o varias tareas cotidianas diariamente. Aproximadamente el 10% de la población mayor de 65 años tiene problemas cognitivos que a su vez se asocian a problemas funcionales. Por consiguiente, el número de personas con algún tipo de discapacidad se estima cercano al 13% en Europa y del 15% en EEUU.

1.2.- Robótica Asistencial Social[2]

Por todo el mundo, la población de mayor edad, así como de aquellos que presentan algún tipo de discapacidad, han expresado sus preferencias e ilusiones de poder vivir de la manera más independiente posible en sus respectivas comunidades. Habitualmente hacen uso de dispositivos tecnológicos que les permiten realizar sus actividades en el día a día de manera normal. Actualmente ningún gobierno en el mundo puede satisfacer todas las demandas ni proveer de todos los servicios requeridos por este gran segmento de la p oblación. Es por ello que la ingeniería y automatización recogen este testigo, actuando como un servicio social.

1 La información usada para la escritura de los párrafos 1.1 y 1.2 ha sido extraida del libro “The Engineering Handbook of Smart Technology for Aging, Disability and Independence”; Helal, Mokhtari and Abdulrazak; 2008.

2 Las definiciones usadas en el párrafo 1.2 se han obtenido de “Defining Socially Assistive Robotics”, David Feil-Seifer and Maja J. Mataric, Interaction Laboratory, University of Southern California. In proceedings of the 2005 IEEE 9th International Conference on Rehabilitation Robotics June 28 – July 1, 2005, Chicago, IL, USA.

La Robótica Asistencial Social (o SAR por sus siglas en inglés) es una respuesta a estos problemas. Los robots SAR se definen como una intersección entre la Robótica Asistida (o AR, en inglés), aquellos robots que asisten a las personas con discapacidades físicas a través de una interacción física, y la Robótica Social Interactiva (o SIR), robots cuya principal tarea es la de realizar algún tipo de interacción social Humano-Robot. Los robots SAR comparten con los robots AR la meta de proporcionar asistencia a usuarios humanos, pero, al igual que los SIR, esta asistencia se hace a través de interactuar socialmente. Para los SAR, la meta es el desarrollo efectivo y cercano de la interacción con el usuario humano con el propósito de proporcionarle una asistencia significativa en la rehabilitación, aprendizaje, convalecencia etc.

Las principales características de la SAR son las siguientes:
- Fiabilidad y seguridad: Los factores primordiales en la interacción con humanos. Los errores no son tolerables, porque pueden causar daños a sus usuarios.
- Interfaz de usuario: Sencilla, accesible y personalizada.
- Infraestructura mecánica ergonómica: Deben ser fácilmente trasladables.
- Hardware y Software: Sistemas operativos en tiempo real, algoritmos y arquitecturas de software diseñadas para controlar la estructura metálica adecuadamente.
- Además de todas estas características, la aceptación de los robots y dispositivos SAR es el factor más importante de todos. Deben ser fáciles de usar, no intrusivos y sentirse parte del entorno.

Este proyecto trabaja sobre el campo de los robots SAR. Como puede imaginarse, hay una gran cantidad de posibles desarrollos y avances tecnológicos en este campo, y futuros avances e investigaciones permitirán desarrollar aún más.

1.3.- Automática, tecnología y asistencia para ponerse en pie/levantarse

Uno de los problemas que habitualmente sufren las personas de avanzada edad es la de levantarse cuando están sentados o tumbados. Este proyecto es el primero centrado en solventar dicho problema: Ofrecer asistencia a personas ancianas o con discapacidades físicas que tienen dificultades para ponerse en pie.

Este proyecto ya fue comenzado por Teemu Kuusisto, como parte de su Master's Thesis en el departamento de Automática y Sistemas de la Helsinki University of Technology. Por tanto, en el momento de iniciar éste proyecto, un primer prototipo ya estaba diseñado y construido, pero su software inacabado. Por tanto, el propósito principal de este proyecto es el de mejorar el código del software del prototipo, solucionando errores y añadiendo funcionalidades al mismo. Tras el trabajo realizado en este proyecto, el prototipo debería estar un paso más cerca de poder ser un producto acabado que, con futuros desarrollos, especialmente en la etapa de diseño, podría ser usado en un entorno real.

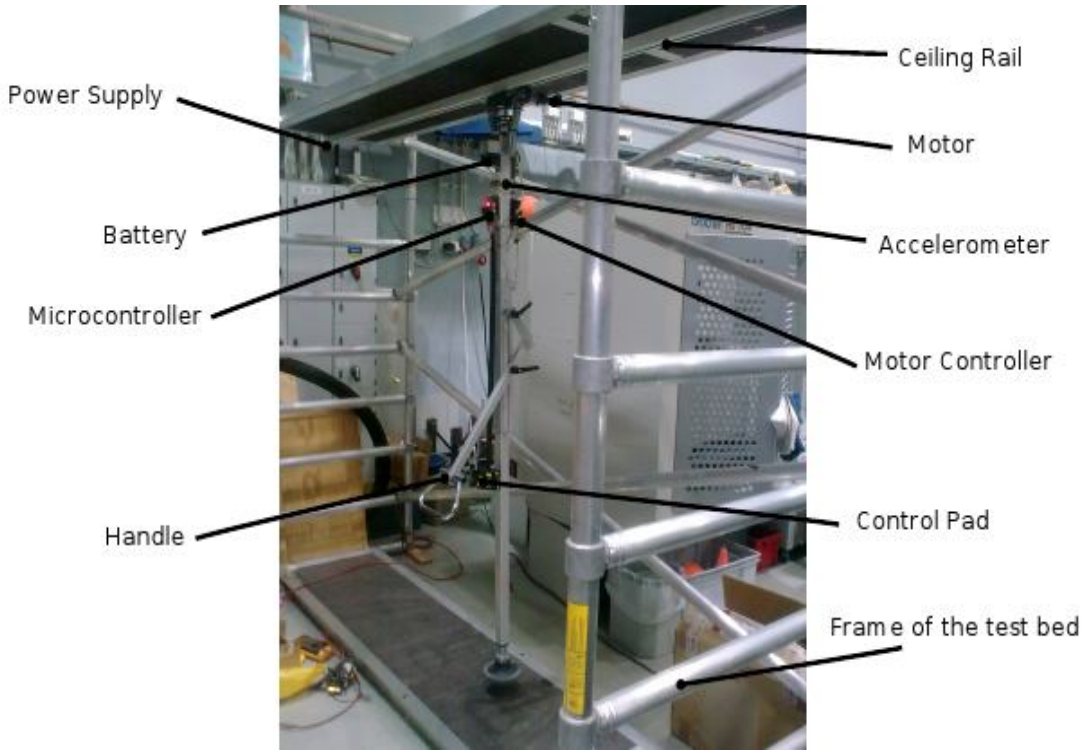
2. Descripción del prototipo de soporte robótico

En este Proyecto Fin de Carrera se realizan mejoras sobre un sistema de soporte robótico no finalizado. Este prototipo fue diseñado por Teemu Kuusisto en su PFC. Por ello, antes de empezar a detallar las mejoras que se han realizado sobre el modelo original, se hace necesario detallar las características constructivas y funcionales del mismo.

2.1.- Esquema del prototipo

El soporte robótico original es básicamente una barra con una manila que se mueve a lo largo del techo siguiendo un camino fijado a lo largo de un raíl. Esta barra es retractile, lo que le permite alargarse y acortarse en un determinado punto de su trayectoria, tensándose y destensándose entre el techo y el suelo. De esta manera, cuando está extendida, actúa como un soporte robusto que puede se puede usar para levantarse, y cuando se desplaza, puede usarse también como soporte y guía en el movimiento.

Inicialmente, el movimiento por el techo se hace a través de un raíl recto, y las órdenes de control se ejecutan a través de un mando de control unido al soporte. El prototipo, así como sus diferentes partes, pueden verse en la siguiente figura (2.1):



(2.1) Soporte robótico sobre estructura de pruebas

2.2.- Descripción del hardware

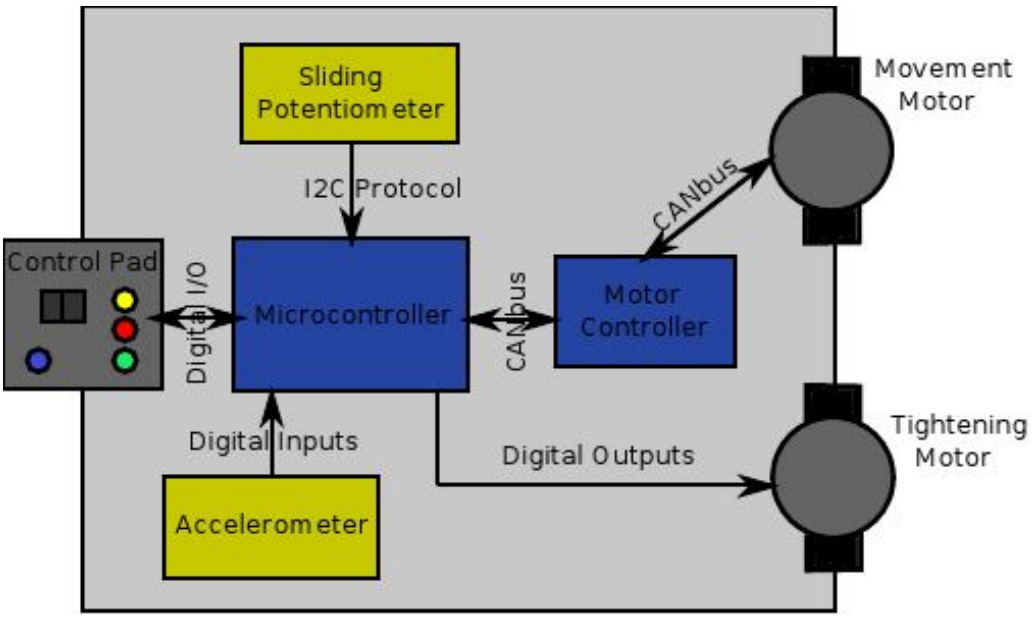
En esta sección se revisar los principales componentes usados para la construcción del prototipo, con el objetivo de poder comprender mejor su funcionamiento.

2.2.1.- Barra del soporte y estructura de pruebas

Tal y como se muestra en la figura (2.1), la estructura en forma de pórtico está diseñada para soportar el peso de la barra, así como de hacerla estable. Tiene un raíl a lo largo de su parte superior, por el cual el soporte se desplaza. Sobre este raíl se han definido arbitrariamente 3 posiciones virtuales, que se corresponden con “Casa” (izquierda de la estructura), “Cama” (posición central) y “Silla” (derecha).

2.2.2.- Motores

Todos los movimientos realizados por el soporte son provocados por la acción de 2 motores independientes de corriente continua. Uno de los motores mueve la barra a lo largo del raíl, mientras que el otro se encarga de su extensión (y por tanto, tensión) y acortamiento (relajación). El motor que mueve la barra hacia la izquierda/derecha a través del raíl está controlado y monitorizado por el Módulo de Control Inteligente PIM3605 de Technosoft, comunicándose a través de CANbus. La tensión de la barra es controlada de manera directa por el microcontrolador central utilizando sus salidas digitales. Puede encontrarse más información sobre éste controlador en su hoja de características [1] y guía del usuario [2].



(2.2) Esquema de las comunicaciones y hardware del prototipo

2.2.3.- Microcontrolador AT90CAN128 de Atmel

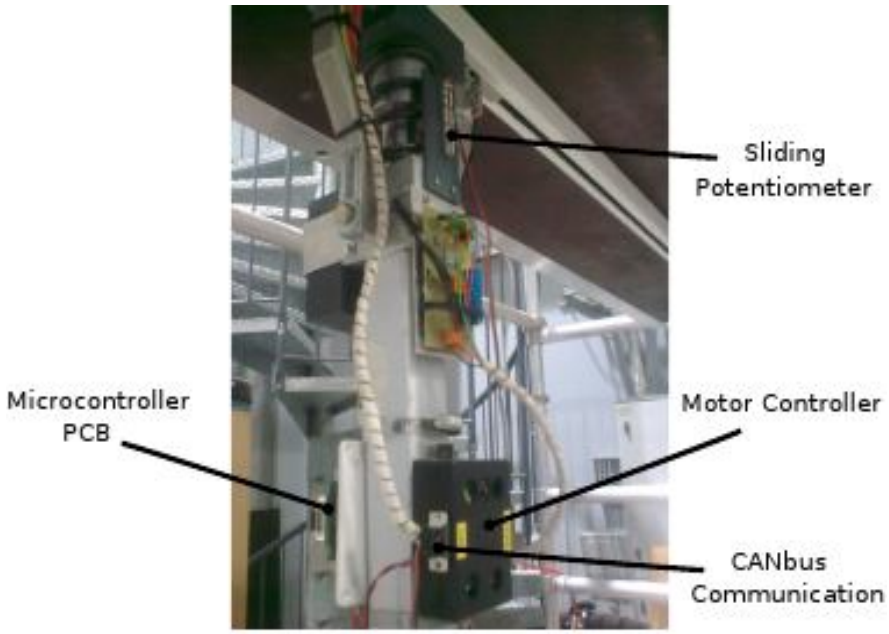
El AT90CAN128 es un microcontrolador de 8 bits de uso general con diversas implementaciones y añadidos que permiten mejorar sus capacidades de almacenamiento y comunicación. Se puede encontrar información más extensa sobre el microcontrolador en su hoja de características [3].

Todos los cálculos y tareas de control del prototipo, excepto aquellas relacionadas con el motor de desplazamiento por el raíl, son realizados en el microcontrolador. El software de control implementado es principalmente una máquina de estados actualizada a través de interrupciones que controla los estados de las entradas, salidas y estados del soporte robótico. Proporciona las órdenes adecuadas al motor de acortamiento/extensión y hace los cálculos y conversiones A/D requeridas para ello. Además, se comunica con el resto de elementos del sistema de diferentes modos y usando diferentes protocolos (Ver figura (2.2)):

- CANbus para la comunicación con el controlador del motor.
- Comunicación en serie a través de 2 UARTs para la conexión con el PC (para la carga de los datos en la ROM y para tareas de supervisión)
- Bus I2C para la comunicación con el acelerómetro.
- Puestos I/O digitales para la comunicación con el mando de control y el potenciómetro deslizante.

2.2.4.- Acelerómetro LIS3L02

El microchip LIS3L02 es un acelerómetro de 3 ejes usado para medir la inclinación de la barra del soporte. Los valores de cada uno de los 3 ejes son números enteros de 12 bits (y por tanto, con valores entre 0 y 4095). El estándar de comunicación entre el acelerómetro y el microcontrolador es el protocolo I2C. Puede encontrarse más información en su hoja de características [3].

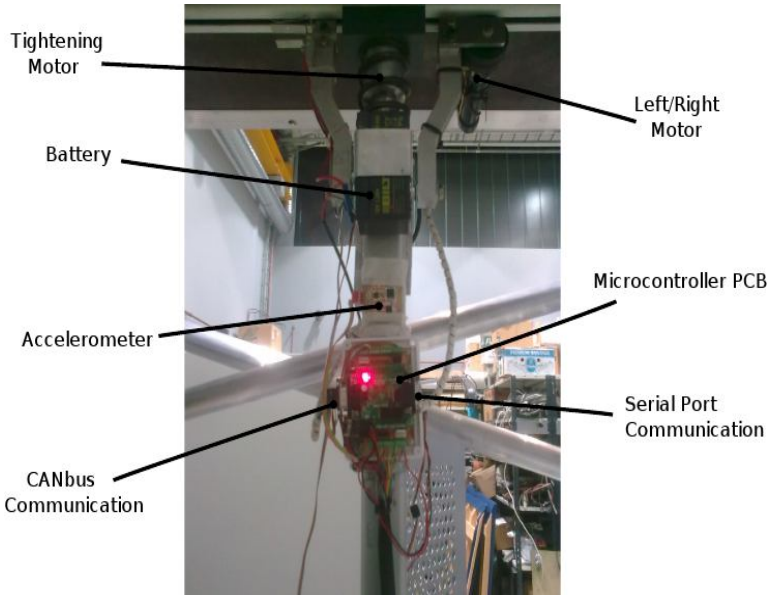


(2.3) Detalle de la parte superior trasera del soporte robótico

2.2.5.- Potenciómetro deslizante

El soporte puede extenderse y contraerse, siendo capaz, por tanto, de quedarse fijado entre el suelo y el techo en cualquier parte de su trayectoria. Cuando el soporte se extiende hasta quedar totalmente tenso, debe asegurarse de que el valor de dicha tensión es lo suficientemente elevado como para que una persona pueda apoyarse en él con seguridad. Es aquí donde entra en juego el potenciómetro deslizante, del que se lee continuamente su resistencia (lineal en relación al desplazamiento vertical del soporte). Cuando el valor de la misma supera cierto valor umbral, establecido experimentalmente, el motor se detendrá, pues el soporte estará lo suficientemente tenso.

Puesto que los valores proporcionados por el potenciómetro deslizante son analógicos, es necesario un circuito de acondicionamiento y amplificación, así como un conversor A/D para que el microcontrolador pueda leer este valor a través de uno de sus puertos digitales.



(2.4) Detalle de la parte superior delantera del soporte

2.2.6.- Mando de control

El mando de control es usado para proporcionar las instrucciones binarias adecuadas para poder hacer funcionar al soporte. Estas órdenes son leídas por el microcontrolador, que en respuesta manda las señales adecuadas al controlador del motor. Cada uno de los botones del mando de control está conectado a un pin de uno de los puertos I/O digitales del microcontrolador.

Botones usados en el mando de control (ver figuras (2.5), (2.6) y (2.7)):
- Interruptor On/Off. Enciende o apaga el dispositivo.
- Interruptor Auto/Man: Selector de modo entre los modos manual y automático.
- Botón Vasen/Oikea (Left/Right): Movimiento a izquierda/derecha en el modo manual.
- Koti (Home): Movimiento hasta la posición “Home” en el modo automático
- Sänky (Bed): Movimiento hasta la posición “Bed” en el modo automático

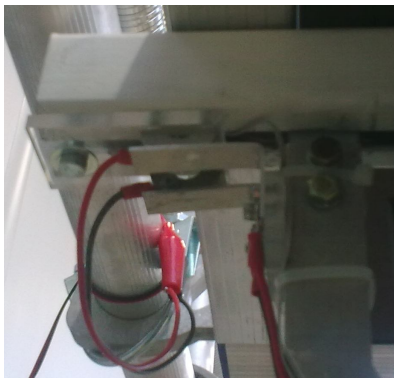
- Tuoli (Chair): Movimiento hasta la posición “Chair” en el modo automático
- Kiristys (Tightening): Tensión/Extensión en cualquiera de los dos modos

El mando de control tiene también 2 LEDs (ver figura (2.6)) controlados mediante salidas digitales del microcontrolador:
LED rojo: LED de espera, encendido cuando el sistema espera una instrucción.
LED amarillo: LED de acción, encendido cuando el sistema ejecuta una acción.

(2.5), (2.6) y (2.7) Mando de control: Vistas laterales y frontal

2.2.7.- Batería y fuente de alimentación

El soporte es alimentado por una bacteria de 12V adosada a la barra. Esta batería se recarga cuando el soporte se halla en la posición “Home”. Esto es posible gracias a la acción de dos pines a una diferencia de potencial de 12V DC localizados al final del raíl. Cuando el soporte se encuentra en la posición “Home”, hace contacto con dichos pines (ver figuras (2.8) y (2.9)). Es muy importante, entonces, dejar siempre el soporte en la posición home al apagarlo para evitar una posible descarga total de la batería.



(2.8) Recargando (Posición “Home”)

(2.9) Descargando

3. Pruebas iniciales con el soporte robótico

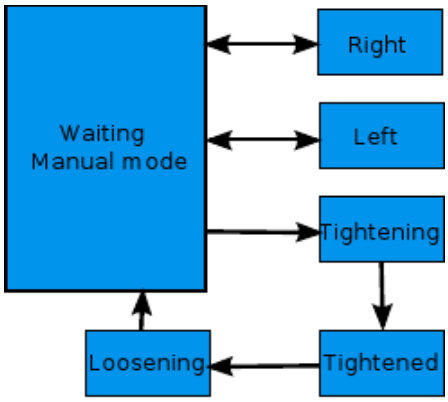
Tras estudiar los fundamentos de la estructura del prototipo de trabajo y su hardware, se realizaron las primeras pruebas para comprobar su funcionamiento. El propósito de estas primeras pruebas, por tanto, es el de determinar las funcionalidades, capacidades, limitaciones y errores de diseño del prototipo. Previamente a la realización de estas pruebas, se hace necesario el estudio minucioso del código original implementado en el microcontrolador [5]. Como puede verse en el propio mando de control, el prototipo puede alternar entre dos modos principales: Automático y Manual. Cada uno de estos dos modos determina las acciones a realizar por el prototipo y los botones del mando que podrán usarse.

3.1.- Modo manual

Botones usados: “Left”, “Right” y “Tightening”.

Mientras son pulsados, los botones “Left” y “Right” mueven el soporte hacia la izquierda y derecha sobre el raíl a una velocidad fija.

El botón “Tightening” tenía dos usos: Cuando el soporte estaba destensado, daba las órdenes necesarias al controlador del motor de extender la barra hasta que ésta estaba tensa y firme entre el suelo y el techo. La detección de la tensión se hace a través de la lectura de la resistencia de un potenciómetro deslizante adosado a la barra, mediante una conversión A/D. Si el soporte se encontraba ya totalmente tenso, apretar el botón lo devolvía a su posición inicial. Durante estos movimientos de tensión/destensión, cualquier instrucción adicional es ignorada, no pudiéndose detener el movimiento. Una vez el soporte se hallaba totalmente tenso, la única instrucción permitida es la de destensarse, cualquier otra instrucción es ignorada.



(3.1) Diagrama de flujo del modo manual en el prototipo

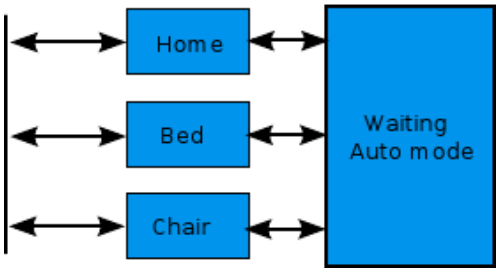
3.2.- Modo automático

Botones usados: “Home”, “Bed”, “Chair” y “Tightening”.

Los botones “Home”, “Bed” y “Chair” se usan para mover el soporte a los puntos definidos por los estados “Home”, “Bed” y “Chair”. El movimiento solo puede detenerse se si cambia al modo manual, pero no en el modo automático. Por tanto, mientras el soporte se desplaza hacia la posición

marcada, ninguna otra acción se llevará a cabo. Estas posiciones se han definido en el programa para estar a la izquierda (“Home”), centro (“Bed”) y derecha (“Chair”) del raíl.

El botón “Tightening” se suponía debiera funcionar de manera similar al modo automático, pero no era el caso. Esto creaba la problemática adicional de no ser capaces de control el sistema en modo automático si estaba el soporte previamente tensado en el modo manual.



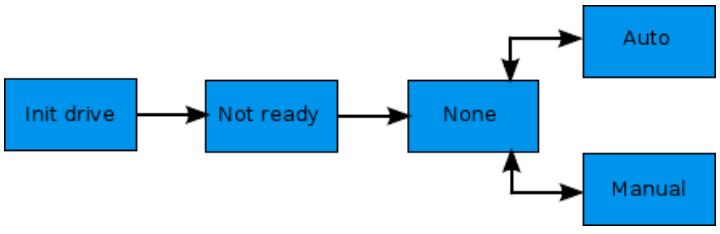
(3.2) Diagrama de flujo del modo automático del prototipo

3.3.- Modos manual y automático:

Ambos modos trabajan en conjunción con el acelerómetro: Durante el movimiento, la tensión, o la intención de llevarlos a cabo, el sistema lee continuamente los valores del acelerómetro, y si éstos superan los valores predefinidos para el movimiento, las instrucciones son ignoradas y los movimientos no son realizados. El propósito de esto es que el soporte solo se mueva/tense cuando su posición sea la vertical, y que por tanto no esté inclinado en dirección alguna. Esta funcionalidad, pese a estar implementada en el código, no funcionaba correctamente. El soporte podía estar inclinado en cualquier dirección, incluso si era agitado violentamente.

Otro defecto presente en el prototipo era la asimetría en su movimiento: La velocidad a la que se tensa/destensa y se mueve hacia la izquierda/derecha no es la misma. Adicionalmente, cuando la barra se contrae, destensando el soporte, no alcanza la posición original, quedándose ligeramente más extendida cada vez. En cualquier caso, este fallo parece ser más un defecto del propio motor que de la programación del prototipo, como pruebas posteriores permitieron determinar.

Es preciso mencionar también la rutina de inicialización implementada. Cuando el dispositivo se enciende, no se acepta ninguna orden hasta que el soporte está totalmente contraído/destensado y en la posición “home”. Esta rutina ha sido probada varias veces y funciona correctamente.



(3.3) Diagrama de flujo de la rutina de inicialización del prototipo

4. Ideas para la mejora del sistema

El prototipo y su estructura son totalmente funcionales, pero hay un gran margen para la mejora y adición de funcionalidades, que de hecho deben ser implementadas para poder cumplir con las expectativas. Esto es, todavía se requiere una gran cantidad de trabajo para que pueda suponer una ayuda real a personas de avanzada edad o con discapacidad física. No obstante, la mayor parte de estas mejoras están fuera del alcance de este proyecto. Las mejoras realizadas en este proyecto se hacen sobre el hardware disponible, tan solo con alguna pequeña adición en forma de módulos añadidos al prototipo original, pero sin modificar en ningún caso la estructura básica del mismo.

A continuación se detallan ideas sobre cómo podría mejorarse el sistema, que más tarde dividiremos en 2 categorías, aquellas mejoras que se realizan en este proyecto y aquellas que no.

- Cambios en el código original y mejoras sobre el mismo
- Hacer funcionar el acelerómetro adecuadamente
- Detección de colisiones básica
- Control manual de la tensión del soporte
- Reconocimiento de voz
- Control asistido básico
- Sistema de movimiento mejorado
- Computación distribuida
- Capacidad de giro
- Control asistido mejorado
- Brazo robótico o asistido

4.1.- Mejoras a realizar en este proyecto

Cambios en el código original y mejoras sobre el mismo

La escritura de tal cantidad de código sin errores es una tarea realmente complicada para una sóla persona. Habitualmente el código creado tiene un gran potencial de optimización y mejoras en su funcionamiento que un programador ajeno al código inicial puede detectar de una manera relativamente sencilla. Por ello, a la vez que he trabajado sobre otras mejoras, partes del código se han cambiado, así como algunas estructuras y algoritmos que han sido simplificados, tratando de mantener compatibilidad y coherencia con el resto del sistema.

Hacer funcionar el acelerómetro correctamente

En el uso del acelerómetro es donde se producen la mayor cantidad de errores del sistema, ya que no parece que esté funcionando apropiadamente durante las pruebas iniciales. Este es un problema, por tanto, que debe ser atajado en este proyecto, para que el soporte no trate de tensarse o moverse cuando está inclinado en alguna dirección, ya que puede ser peligroso incluso para el propio soporte.

Detección de colisiones básica

Tal y como se encuentra en prototipo en este punto, el soporte se mueve en el momento en que recibe una orden del mando de control. Este movimiento trataré de llevarse a cabo incluso aunque haya un obstáculo en el camino, lo cual evidentemente puede ser peligroso. Por ello, añadir la capacidad de detección de colisiones al soporte debe ser una prioridad, aunque ésta sea rudimentaria. Si la barra se encuentra con algún obstáculo que le impide el movimiento en su trayectoria, debe detenerse hasta que el obstáculo sea retirado.

Control manual de la tensión del soporte

No estrictamente necesario en el producto final, pero si muy útil para el desarrollo y pruebas del prototipo, es la adición de un control manual de la tensión del soporte. Es decir, la posibilidad de mover el soporte arriba y abajo a petición del usuario. Esto nos permitirá, además, solventar parcialmente el movimiento asimétrico mencionado con anterioridad a la hora de tensar y destensar el soporte.

Reconocimiento de voz

El mando de control es útil de de cara a la realización de pruebas y muy rápido de implementar, pero no es en cualquier caso el sistema de control ideal para el público que finalmente se beneficiará del uso del soporte. La gran mayoría de personas de avanzada edad, así como discapacitados físicos, no se sentirán cómodas lidiando con un mando de control. Se necesitan sistemas de control más fáciles de usar e inocuos que además tengan un mayor rango de usuarios potenciales. Uno de estos sistemas de control es el reconocimiento de voz. La adición de un modulo de reconocimiento de voz hará que el sistema sea más fácil de usar, puesto que los usuarios no tienen porque ver o saber siquiera dónde se encuentra el mando. El objeto es, por tanto, que el soporte pueda ser controlado mediante órdenes vocales exclusivamente si así se desea.

Control asistido básico

Deshacerse del mando de control y desarrollar un interfaz de control más fácil, simple y cómo de usar es uno de los principales objetivos de este proyecto. En conjunción con el reconocimiento de voz, se implementará otro tipo de control que es, además, relativamente fácil de implementar en su forma más rudimentaria: el control asistido. Si el usuario quiere que el soporte se mueva en una dirección, lo único que deberá hacer es empujarlo levemente para que se incline en la dirección deseada. El acelerómetro detectará esta inclinación y los motores del soporte se activarán para que el soporte comience a moverse.

4.2.- Otras mejoras (fuera del alcance de este proyecto)

Sistema de movimiento mejorado

El diseño básico sobre raíles para el prototipo es funcional de cara al desarrollo de otras mejoras y realización de pruebas, pero necesita ser completamente revisado antes de poder ser

implementado de una manera económica, factible y realista en un hogar. Por el momento únicamente funciona en línea recta, pero la habilidad de poder rotar, girar y elegir diferentes trayectorias mediante cruces y bifurcaciones debería ser desarrollada

En cualquier caso, el movimiento sobre raíles es por definición limitado, pues impide el libre movimiento libre del soporte, al deber seguir el camino marcado por el raíl, no siendo posible, por tanto, cubrir la totalidad del espacio en una habitación. Un sistema de movimiento más evolucionado debería implementarse de cara al futuro. Por ejemplo el uso de ruedas entre el suelo y el techo a la vez, que dotaría al soporte de un radio de acción efectivo de 360º (Figura (4.1)). Desarrollos sobre diferentes aproximaciones a esta tarea (el movimiento de robots por el techo) se están realizando en estos momentos en el departamento de Automatización y Sistema en el proyecto Ceilbot[3].

Computación distribuida

El prototipo funciona actualmente de manera independiente, pero cobraría mayor sentido en una casa totalmente domótica. En este tipo de hogares, un ordenador central con un potente procesador de propósito general realizaría y coordinaría toda la electrónica y comunicaciones. Todos los cálculos no directamente relacionados con el control de movimiento más básico deberían realizarse en el ordenador central y no en el microcontrolador. De este modo, el modulo de reconocimiento de voz funcionaría mejor si se implementase en un ordenador central, y su información procesada de una manera más eficiente. Lo mismo podría decirse para el sistema de localización del soporte, que es ahora realmente simple, pero que debe ser mejorado si el soporte debe moverse por toda una casa de una manera óptima. También permitiría la implementación de mejoras adicionales, como la habilidad de llamar al soporte si éste se encuentra en otra habitación.

Capacidad de giro

En su actual estado de desarrollo, el soporte robótico se mueve sobre el raíl en una posición fija, no puede rotar sobre sí mismo ni sobre el raíl. Esto implica que la manila se encuentra siempre en la misma dirección. Aunque no sea algo totalmente obligatorio, sería realmente útil que se implementara un sistema de movimiento mejorado que dotara al soporte de la capacidad de rotar en un arco completo de 360º. Esta funcionalidad facilitaría en gran medida su facilidad de uso por parte de los usuarios, haciéndolo más cómodo. Con la estructura actual del prototipo esto no puede hacerse, por lo que debería mejorarse. Ver figura (4.2)

Control asistido mejorado

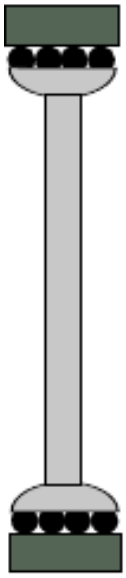
En este proyecto se implementa control asistido en el prototipo en su forma más básica, pero dicho control está muy limitado por su propio diseño. Se está usando un acelerómetro para detector si el soporte está siendo empujado o no. Pese a que esto es funcional, implica que el usuario debe inclinar la barra en la dirección en que desea mover el soporte. Adicionalmente, el mismo acelerómetro se usa para la detección de colisiones, por lo que determinar si la causa de dicha inclinación es debida a un obstáculo en la trayectoria o a que el usuario lo está empujando es ciertamente complejo y no demasiado fiable. Sería mucho más elegante y precisa la adición de

3 Información adicional sobre el proyecto Ceilbot puede encontrarse en http://autsys.tkk.fi/en/Ceilbot

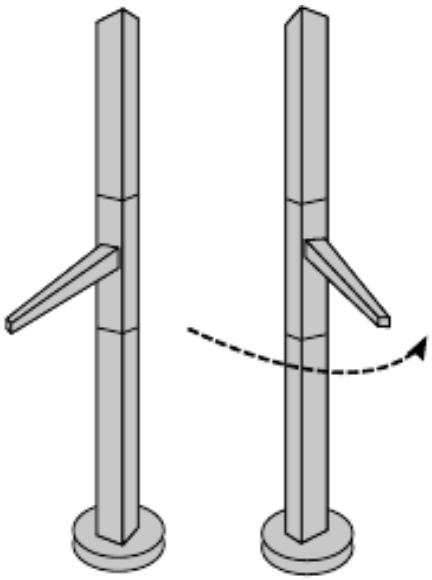
módulos específicos de hardware para esta tarea.

Brazo robótico/asistido

Una vez todos los desarrollos previos hubieran sido implementados, una funcionalidad que realmente ayudaría al público objetivo sería la adición de un brazo robótico o asistido a la estructura del soporte. El soporte está diseñado para ayudar a los usuarios a levantarse y moverse a través de la casa, pero no sirve de gran ayuda si el usuario debe agacharse para coger algo que esté en el suelo. Esta tarea es, además, ciertamente dura para las personas de avanzada edad. Por ello, la adición de un brazo robótico (asistido o no según decisión de diseño) ayudaría a los usuarios en situaciones que realizan diariamente.



(4.1) Alternativa al movimiento por el techo



(4.2) Capacidad de giro

5. Implementación de las mejoras

En esta sección se implementarán todas las mejoras y funcionalidades adicionales listadas en la sección (4.1). Posteriormente todas estas mejoras serán probadas y evaluadas.

A lo largo de esta sección se harán numerosas referencias a ficheros, extractos de código, así como estados y/o funciones y argumentos. Para facilitar la identificación de todas estas referencias, el formato de escritura usado para cada una de ellas será el siguiente:
- Ficheros en cursiva. Ej. *state_machine.c*
- Código en una fuente diferente. Ej. `is_straight (int state)`
- Estados/Argumentos en mayúsculas. Ej. STATE_MOVE_RIGHT

Antes de desarrollar la implementación de las mejoras desarrolladas, es conveniente mostrar un diagrama de flujo que muestre el funcionamiento buscado para el sistema final. Estos diagramas permiten entender de forma más completa los objetivos a desarrollar, viéndolos en su contexto de operación. En estos diagramas de flujo, los estados son representados por rectángulos, mientras que las flechas indican las órdenes que permiten la transición entre diferentes estados.

Rutina de inicialización

Esta rutina ya estaba implementada adecuadamente en el prototipo. El estado “NOT READY” destensa por completo el soporte y lo mueve a la posición “Home”. Durante el estado temporal “NONE” se realice la lectura del interruptor que indica el modo de traba, de modo que el sistema automáticamente tome el estado “AUTO” o “MANUAL”. Finalmente, el sistema quedará a la espera de las siguientes órdenes. Ver figura (3.3)

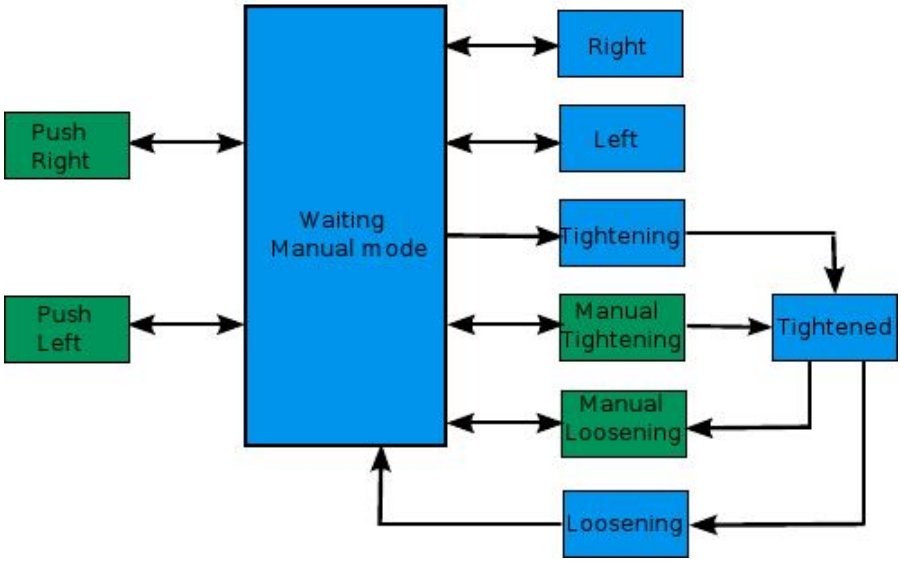
Modo manual

Los estados disponibles desde el modo manual pueden verse en el diagrama de estados de la figura (5.1). Puesto que la mayoría de estados funcionan únicamente mientras determinado botón del mando de control es presionado o el soporte inclinado, se debe retornar siempre hacia el modo de espera manual para que se puedan aceptar instrucciones adicionales. Esto se representa en el diagrama mediante flechas bidireccionales. Aquellos estados añadidos sobre el prototipo original están coloreados en verde, mientras que aquellos que ya estaban presentes se indican en azul.

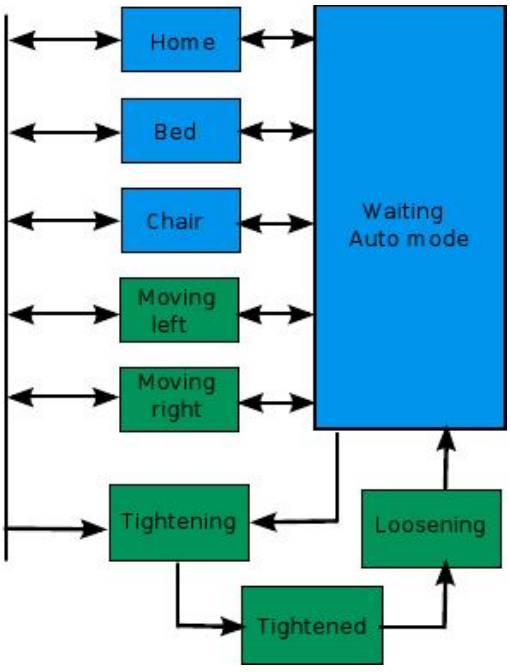
Modo automático

El modo automático es similar al modo manual. La principal diferencia es que las transiciones entre estados no deben pasar por un estado de espera. (Ver figura (5.2)). Todos los estados pueden ser activados utilizando indistintamente el mando de control o el reconocimiento de voz, salvo “MOVING_LEFT” y “MOVING_RIGHT”, que únicamente pueden ser activados por reconocimiento de voz. El esquema de comunicación a modo de bus lineal entre estados indica que el sistema puede pasar de uno de los estados a otro sin pasar por el estado de espera. De manera idéntica al diagrama del modo manual, los estados coloreados en verde son aquellos añadidos sobre el prototipo original, y

los azules aquellos que ya existían previamente.



(5.1) Diagrama de flujo del modo manual



(5.2) Diagrama de flujo del modo automático

5.1.- Optimización y mejoras sobre el código original

Antes de la adición de módulos de hardware adicionales sobre el prototipo, se realizaron ciertos cambios y depuraciones sobre el código original. Dichos cambios se describen en esta sección y se listan por orden cronológico: Cuando un cambio se indica después de otro, es que tuvo lugar más adelante en el tiempo.

NOTA: La rutina principal que gestiona el soporte ese trata básicamente de una máquina de estados dirigida por interrupciones, siendo programada la máquina de estados en los ficheros *statemachine.c* y *statemachine.h* y las interrupciones en *interrupts.c* y *interrupts.h*. La máquina de estados tiene definidos estados para todas y cada una de las acciones que el soporte puede realizar. Es por ello que numerosas veces en esta sección se hará referencia a dicha máquina de estados y sus estados.

5.1.1.- Preparación del código original

Tras estudiar en profundidad el código, y con objeto de familiarizarnos con él, comenzamos con 3 tareas simples. Primero, renombrando los ficheros, del idioma Finés al Inglés; Segundo, con un borrado de código redundante; y tercero, reescribiendo la función de tensado/destensado para el modo manual.

5.1.1.1.- Renombrado de ficheros

Muchos de los ficheros originales ya tenían nombres en ingles, puesto que son extractos de código fuente con licencia GPL que se usan directamente para controlar y configurar el hardware del sistema. No obstante, todos aquellos ficheros escritos por Teemu Kuusisto, el creador del prototipo, tienen nombres en finés. A continuación se incluye la lista de ficheros renombrados:

tanko.c → *main.c*
napit.c → *statemachine.c*
napit.h → *statemachine.h*
moottori.c → *motor.c*
moottori.h → *motor.h*
ledit.c → *leds.c*
ledit.h → *leds.h*
keskeytys.c → *interrupts.c*
keskeytys.h → *interrupts.h*
liukupotentiometri.c → *slidepotentiometer.c*
liukupotentiometri.h → *slidepotentiometer.h*
summeri.c → *buzzer.c*
summeri.h → *buzzer.h*

El fichero *Makefile* también se cambia en concordancia.

5.1.1.2.- Borrado de código redundante

El código original utilizaba dos ficheros de código C diferentes para el control del acelerómetro, con sus respectivos ficheros de cabecera asociados. Puesto que esto no es necesario, dichos ficheros fueron integrados en uno sólo. En consecuencia, los ficheros *accelerometer.c* y *kiihtyvyysanturi.c* pasan a formar parte de *accelerometer.c* (igualmente para sus ficheros de cabecera). Esto nos genera código redundante, con funciones de nombres diferentes pero mismo propósito, por lo que son borradas, y las llamadas a las mismas revisadas.

5.1.1.3.- Tensión en el modo automático

Ésta funcionalidad estaba ausente en el prototipo original. Los estados que gestionan la tensión/destensión del soporte (STATE_TIGHTENING, STATE_TIGHTENED y STATE_LOOSENING) estaban preparados para trabajar en los estados manual y automático, pero no sucedía lo mismo con los estados que gestionaban el modo automático (STATE_AUTO, STATE_HOME, STATE_BED y STATE_CHAIR). La solución fue por tanto la de añadir código adicional en estos estados de manera similar al que ya existía para los estados del modo manual.

5.1.2.- Control manual de la tensión

La adición del control manual de la tensión requería ya algo más de trabajo.
El propósito era el de dotar al soporte de la habilidad de moverse hacia arriba y abajo cuando se daban las órdenes apropiadas, tensando y destensando el soporte manualmente. Puesto que el mando de control ya no tenía más botones disponibles, se usaron para este propósito los botones “Home” y “Bed” como si fueran los botones “manual tightening” y “manual loosening”. Estos botones no eran usados en el modo manual, únicamente en el automático, por lo que no se genera en el proceso ninguna incompatibilidad u error. La posición relativa de dichos botones en el mando nos permite recordar su función.

La máquina de estados que gobierna el sistema ya tiene definidos estados para la tensión y destensión, pero éstos ignoran cualquier orden mientras llevan sus acciones a cabo. Por ello dos nuevos estados han sido creados: STATE_MANUAL_TIGHTENING y STATE_MANUAL_LOOSENING.

En STATE_MANUAL_TIGHTENING la barra se extiende, moviéndose hacia abajo (tensando el soporte) mientras el botón “manual tightening” esté presionado. En STATE_MANUAL_LOOSENING se realice la acción contraria, moviéndose la barra hacia arriba (destensando el soporte) mientras el botón “manual loosening” se mantenga presionado. Estos estados son similares a los estados “STATE_LEFT” y “STATE_RIGHT” del modo manual, que mueven el soporte hacia la izquierda/derecha cuando el botón apropiado es presionado.

Para prevenir accidentes o un funcionamiento erróneo, se hace necesario leer los valores del potenciómetro deslizante a la vez que el soporte está tensándose manualmente, de manera similar al estado en que se tensa automáticamente (“STATE_TIGHTENING”). No hay necesidad de añadir un límite cuando la barra se contrae, puesto que cuando el límite superior es alcanzado, el motor se detiene automáticamente.

Es importante recalcar que en este punto del proyecto, es decir, el momento en que el control manual ha sido implementado con el objeto de poder realizar pruebas y ganar familiaridad con el código del sistema, el acelerómetro aún no funciona correctamente. Por tanto, es todavía posible tensar el soporte aunque no se encuentre en posición totalmente vertical. Esta problemática será solucionada en la siguiente sección.

5.1.3.- Pruebas con el acelerómetro: Funcionalidad básica

Durante la realización de las primeras pruebas, quedó claro que el acelerómetro no estaba funcionando apropiadamente. La barra podía moverse violentamente en cualquier dirección sin afectar en modo algún el funcionamiento del soporte. Tras examinar el código y realizar algunas pruebas imprimiendo por pantalla los valores de los ejes “X”, “Y” y “Z” del acelerómetro, la causa del error salió a la luz: No estaba funcionando dados los valores umbral excesivamente permisivos definidos en la función que controlan el acelerómetro.
Ésta función, `is_straight()`, implementada en el código original, determina si el soporte está lo suficientemente erguido (vertical) para moverse o tensarse. Si los valores leídos desde el acelerómetro sobrepasan los límites definidos para la función `is_straight()`, cualquier instrucción de movimiento será ignorada.

Por tanto, la impresión por pantalla de los valores de los 3 ejes del acelerómetro y la comprobación de los estados que alcanza la función `is_straight()`, así como pruebas iterativas, permitieron solucionar el problema. Tras esta mejora, si el soporte es inclinado levemente, aproximadamente 3 grados, se detendrá. Los límites definidos en este punto han sido definidos a propósito de manera que restringen en gran medida el movimiento del soporte, puesto que su objetivo es trabajar en conjunción con desarrollos posteriores sobre el control del acelerómetro que se detallan en las siguientes secciones.

NOTA: Es preciso indicar que evidentemente los valores proporcionados por el acelerómetro dependen de la orientación del mismo sobre el soporte. Si el modulo de hardware del acelerómetro se desplaza hacia otra parte de soporte, los valores deberán ser reconfigurados.

5.1.4.- Mejoras sobre el acelerómetro: Detección de colisiones

La rutina de control del acelerómetro implementada en la sección anterior está diseñada en realidad para un soporte estático, es decir, cuando el soporte no está moviéndose y recibe alguna instrucción de movimiento. Cuando el soporte se está moviendo, los valores leídos desde el acelerómetro no son los mismos, por lo que los estrictos límites definidos para el soporte estático no son adecuados, deteniendo continuamente el soporte. Es necesario, por tanto, mejorar la función `is_straight()`, tal que pueda identificar si el soporte se mueve hacia la izquierda o derecha. Adicionalmente, a la vez que realizamos estos cambios, se implementa una detección de colisiones rudimentaria.

El objetivo es, por tanto, que el soporte se mueva conociendo la dirección que esta siguiente, y que si se encuentra con algún obstáculo en su trayectoria que le impida su movimiento, se detenga, evitando posibles problemas y/o daños. Esto se hace posible porque cuando la barra entra en contacto con un obstáculo, tratará de seguir su movimiento, por lo que se inclinará en dirección al sentido de su movimiento. El acelerómetro detectará esta inclinación y la función `is_straight()` evitará que el soporte continúe su movimiento. Los siguientes cambios de código tratan de implementar la detección de colisiones tanto en el modo manual como en el automático.

Los valores límite definidos para el acelerómetro cuando el soporte se desplaza serán asimétrico, dependiendo de la dirección del movimiento, ya sea izquierda o derecha. Por tanto, cambios en el código y definiciones adicionales son requeridos para la función `is_straight()`. Estos cambios son diferentes para los modos automático y manual.

El primer paso que se tomó fue el de leer los valores del acelerómetro cuando el soporte se mueve hacia la izquierda/derecha en los dos modos. Éstos valores son diferentes dependiendo del sentido del movimiento, pero independientes, como es lógico, de que el modo sea manual o automático. Con estos nuevos valores se definen nuevos límites para la función `is_straight()` cuando el soporte se mueve en alguna de las dos direcciones. Por tanto, la función debe conocer también el sentido del movimiento, ya sea izquierda, derecha o estático, para poder asignar los límites adecuados para el acelerómetro.

Necesitamos por tanto que `is_straight()` lea el tipo de movimiento del sistema. Para conseguir esto, la función leerá un argumento que le proporcionará la máquina de estados. Será, por tanto:

```
is_straight (int state)
```

Siendo los definidos los valores para “state” como:

1: Moviéndose hacia la izquierda (state = MOVING_LEFT)
2: Moviéndose hacia la derecha (state = MOVING_RIGHT)
0: Resto de situaciones (state = STATIC)

Esto es algo realmente fácil de implementar en el modo manual, puesto que todos los estados llamarán a la función `is_straight(STATIC)`, salvo “STATE_LEFT”, y “STATE_RIGHT” que llamarán a las funciones `is_straight(MOVING_LEFT)` y `is_straight(MOVING_RIGHT)`.

El modo automático es, en cambio, algo más complejo. Puesto que las posiciones “Home”,

“Bed” y “Chair” están definidas, sabemos que cuando el botón “Home” sea presionado, el soporte se moverá hacia la izquierda independientemente de su posición original (luego realizará la llamada a la función `is_straight(MOVING_LEFT)`). También sabemos que cuando se presiones “Chair”, el soporte se moverá hacia la izquierda (llamando a la función `is_straight(MOVING_RIGHT)`) pero el movimiento cuando el soporte se encuentra en la posición “Bed” no es tan simple. Por tanto, para dicha posición es necesario implementar un algoritmo que identifique la dirección del movimiento.

Para una correcta llamada a `is_straight(movement)`, el estado del sistema es almacenado en una variable adicional. Este estado no es el mismo que el de la máquina de estados, su no uno especial que almacena el valor de algunos estados previos. Por ejemplo, cuando el soporte vaya desde la posición “Home” hasta “Bed”, el valor almacenado será “STATE_HOME”, y cuando vaya de “Chair” a “Bed”, el valor será “STATE_CHAIR”. Por tanto, comparando el valor “STATE_BED” con el estado previo, se puede identificar el sentido del movimiento. Este estado no se salva cuando el soporte está tensándose o destensándose, por lo que el valor será siempre “STATE_CHAIR” o “STATE_HOME”, pero se reinicializará si cambiamos al modo manual.

Estas modificaciones en el código logran hacer funcionar correctamente las rutinas de detección de colisiones. Es necesario aclarar, no obstante, que dada la elevada sensibilidad del acelerómetro, los valores leídos por el mismo cambian constantemente. Por ello, los límites establecidos para la detección de colisiones no pueden ser demasiado estrictos, por lo que el soporte sólo detecta obstáculos si se inclina algo más de 2 ó 3 grados, momento en el que detiene su movimiento.

5.1.5.- Desarrollos adicionales con el acelerómetro: Control asistido

El objetivo es ahora la implementación de un control asistido en su forma más básica. Esto es, si el soporte es inclinado en una determinada dirección por el usuario, no será necesario el uso del mando de control, puesto que el soporte se moverá automáticamente en la dirección inclinada. Lógicamente, esta funcionalidad sólo será implementada en el modo manual.

Puesto que el prototipo no dispone de ningún sensor que le permita detectar si está siendo empujado o sometido a alguna presión, esta funcionalidad está basada en una modificación del algoritmo de detección de obstáculos. Si el soporte se encuentra en el modo manual y está parado, leerá constantemente el eje 'Y' del acelerómetro, paralelo al eje de la barra. Si detecta una inclinación apreciable del eje, procederá a dar las órdenes adecuadas al motor para que mueva al soporte en la dirección indicada. Es similar a la modificación anterior, pero con un ligero cambio en los valores umbral del acelerómetro.

Dos estados nuevos se añaden a la máquina de estados, "STATE_PUSH_LEFT" y "STATE_PUSH_RIGHT". Únicamente podrán ser alcanzados si en el modo manual los límites definidos para un soporte erguido y estático son sobrepasados. Sólo si los valores del eje "Y" detectan una variación significativa, el sistema alcanzará uno de estos dos estados.

Estos dos estados llamarán entonces a las funciones `is_straight(PUSHING_LEFT)` o `is_straight(PUSHING_RIGHT)`, que tienen definidos valores umbral diferentes para el acelerómetro. Mientras los valores leídos por el acelerómetro se encuentren dentro de estos límites, el soporte continuará su movimiento. Estos valores umbral se han obtenido mediante pruebas de carácter iterativo, para que el soporte se mueva únicamente si es empujado.

Un efecto colateral que aparece tras la implementación de esta funcionalidad, es que mejora la operación de la detección de colisiones. Esto es así porque siempre que un obstáculo es detectado, el soporte se detiene, en este momento determinará que está siendo empujado por el obstáculo y por tanto se moverá en dirección opuesta al mismo. Esta acción no lo separará totalmente del obstáculo, pero sí que disminuirá la carga sobre el mismo.

5.2.- Reconocimiento de voz

La implementación de reconocimiento de voz al sistema se corresponde con la parte más compleja de todas las mejoras realizadas sobre el sistema.

5.2.1.- Elección del modulo de reconocimiento de voz

Tras un estudio detallado sobre la materia, quedó claro que el microcontrolador AT90CAN128 no era capaz de realizar la tarea. El reconocimiento de voz es bastante exigente en términos de capacidad de procesamiento y el chip ya está actualmente saturado a la hora de realizar algunas tareas (por ejemplo en la conversión A/D para el uso del potenciómetro deslizante). Por ello, debe añadirse un modulo adicional que realice estos cálculos. Antes de elegir ningún componente, debemos hacer un pequeño estudiar las características que debe tener el módulo añadido, así como las limitaciones impuestas por nuestro sistema.

El módulo debe ser capaz de sustituir el mando de control, por tanto, sólo necesita reconocer unas pocas palabras, no muchas más que botones en el mando. En consecuencia, no se requiere un moderno, complejo y caro sistema de reconocimiento de voz. Una solución simple y barata debería servir.

Una vez definidas las características requeridas por el modulo, debemos considerar las limitaciones que tenemos a la hora de implementarlo en nuestro sistema. Estas limitaciones se reducen prácticamente al abanico de conexiones disponibles que nos permite nuestro microcontrolador. Además de sus puestos I/O digitales, el microcontrolador AT90CAN128 tiene 2 puertos serie, uno de los cuales se usa para su programación y pruebas, pero el otro queda accesible. Por tanto, la conexión puede hacerse bien sea usando las entradas y salidas digitales de los puertos del microcontrolador, o bien mediante comunicación en serie.

Tras la búsqueda de información en Internet y obtener información detallada sobre varios productos (pidiendo información sobre productos a las tiendas), se escogió el módulo de reconocimiento de voz “VRbot”, de VeeaR (http://www.vee-ar.com/).

5.2.2.- Características del modulo VRbot

VRbot es un modulo de reconocimiento de voz simple, pequeño y poco costoso diseñado para trabajar con robots, en especial con la serie Robonova. Se comunica con otros dispositivos mediante una UART.

Principales características:

- 26 órdenes integradas independientes del usuario (Speaker Independent commands, SI) preparadas para dar órdenes simples. Actualmente soportan inglés (EEUU), alemán, italiano y japonés.
- Hasta 32 órdenes definidas por el usuario (Speaker Dependet commands, SD). Palabras de disparo, órdenes y contraseñas de voz.
- Interfaz gráfica de usuario fácil de usar para programar las órdenes de voz.
- El módulo puede usarse sobre cualquier sistema con una mediante una interfaz (alimentada entre 3.3 y 5V).
- Protocolo serie para el acceso y programación del modulo, operando entre 9600bps y 115000bps.

Hardware incluido en el paquete:

- VRbot PCB
- Micrófono
- Cables para la comunicación en serie

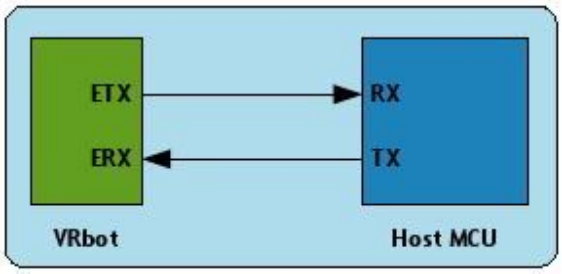
(5.3) Paquete de VRbot

5.2.3.- Conexión de VRbot a un PC

Para comenzar con las pruebas del modulo de VRbot, debe conectarse primeramente a un PC. Puesto que está específicamente diseñado para ser usado con los la serie de robots Robonova, y éstos no se usan en el proyecto, una placa externa genérica de comunicación RS-232 con conector DB9 debe ser usada para su comunicación con un PC. Para la alimentación de ambos módulos deberá usarse una fuente de alimentación de 5V DC. Información detallada sobre la comunicación en serie del módulo puede encontrarse en su hoja de características [7].

(5.4) Diagrama de puertos del modulo de VRbot.

(5.5) Placa externa RS-232



(5.6) Diagrama de conexiones

Parámetros de comunicación a través del Puerto serie:

- Velocidad de transmisión, en BD: 9600 (defecto), 19200, 38700, 57600, 115200
- 8 Bits de datos
- Sin paridad
- 1 bit de parada
- Sin control de flujo

Puesto que el microcontrolador AT90CAN128 tiene por defecto estos mísmos parámetros para la comuniación en seroe, no debemos reconfigurar ni cambiar nada para conectar ambos sistemas. No obstante, estos parámetros deben tenerse en cuenta a la hora de conectar y realizar pruebas con el modulo al conectarlo a un PC.

5.2.4.- Realización de pruebas sobre el modulo con VRbot GUI

Con el modulo apropiadamente conectado al PC a través del Puerto serie, su interfaz gráfica de usuario (o GUI), descargable desde la web del proveedor, ya puede ser usada. Por tanto, abrimos el programa y pulsamos 'connect'. Puede también descargarse un tutorial detallado sobre como enseñar y entrenar órdenes al módulo [8].

El modulo puede reconocer dos tipos de palabras: Órdenes independientes del usuario (SI) y órdenes dependientes del usuario (SD).

Las órdenes SI se corresponden con patrones de reconocimiento de voz integrados en el firmware del dispositivo. A lo largo del tiempo el proveedor irá actualizando dichos patrones, para añadir más órdenes SI y más idiomas. En este momento, existen 25 órdenes SI en 4 lenguajes diferentes: Inglés, alemán, italiano y japonés. En cualquier caso, éste tipo de órdenes no van a ser usadas en el proyecto, ya que están preparadas para el trabajo con los robots Robonova, pero no son adecuadas para nuestro sistema.

Las órdenes SD son en cambio órdenes que deben ser enseñadas al modulo, y son por tanto, independientes del idioma. Éste tipo de órdenes son las usadas en el proyecto.

Tanto las órdenes SI como SD se dividen en dos categorías, “palabras de disparo” y “órdenes”. Las palabras de disparo se usan para preparar al módulo para la recepción de una orden. No son intrínsecamente necesarias, pero sí útiles para evitar el reconocimiento de palabras cuando no se necesitan. El uso habitual de las palabras de disparo es el de tener al módulo tratando sistemáticamente de reconocerlas, para, en caso afirmativo, empezar con el reconocimiento de órdenes. Las palabras de disparo usan también patrones de reconocimiento distinto, ya que el módulo es mucho menos restrictivo en su reconocimiento que para las órdenes.

A la hora de gestionar las instrucciones y órdenes con VRbot GUI, éstas se dividen en “Grupos de palabras” (Wordsets). El grupo de palabras 0 (Wordset 0) se reserve para las palabras de disparo, mientras que los grupos del 1 al 15 se reservan para las órdenes.

A continuación se detallan las palabras que han sido enseñadas al módulo (indicadas en mayúsculas):

Wordset 0:
-INSTRUCTION: Esta es la palabra de disparo. Si el usuario quiere que el módulo reconozca cualquiera de las otras palabras, deberá decir primero esta palabra: “Instruction” para posteriormente pronunciar cualquiera de las otras. Es importante destacar que dadas las limitaciones del módulo, una pausa aproximada de 2 segundos es necesaria entre las dos palabras.

Wordset 1:
-Orden 0: LEFT: Cuando esta orden es recibida, el soporte debería moverse hacia la izquierda, parando únicamente si recibe otra orden o llega al final del raíl.
-Orden 1: RIGHT: Similar a LEFT, pero moviéndose hacia la derecha.

-Orden 2: GO_HOME: Misma función que el botón “Home” en el mando de control.
-Orden 3: GO_TO_BED: Misma función que el botón “Bed” en el mando de control.
-Orden 4: GO_TO_CHAIR: Misma función que el botón “Chair” en el mando de control.
-Orden 5: TIGHTEN: Misma función que el botón “Tighten” en el mando de control.
-Orden 6: STOP: Orden necesaria para la parada del soporte cuando se está moviendo, puesto que las acciones se ejecutan de manera automática hasta su finalización si otra orden no es reconocida.

Tras la enseñanza de estas palabras, se puede verificar su correcta detección utilizando el programa VRbot GUI. Es momento, entonces, de supervisar la el funcionamiento del módulo.

Las órdenes usadas para el trabajo con el dispositivo son órdenes en ingles, pero los grupos de palabras 2 y 3 han sido utilizados para los idiomas finés y español, respectivamente. Estos grupos de palabras han sido también probados. Por tanto, las órdenes enseñadas al modulo finalmente han sido las siguientes:

#	**Wordset 1**	**Wordset 2**	**Wordset 3**
0	Left	Vasemmalle	Izquierda
1	Right	Oikealle	Derecha
2	Go home	Kotiin	A casa
3	Go to bed	Sänkyyn	A la cama
4	Go to chair	Puolille	A la silla
5	Tighten	Kiristys	Tensar
6	Stop	Seis	Para

Durante la realización de este proyecto, solo se ha usado el wordset 1. No obstante, es realmente fácil el uso de los wordsets 2 y 3 también. De hecho, podría añadirse un selector de idioma al sistema.

5.2.5.- Probando y monitorizando el modulo con HyperTerminal

En esta sección se detalla cómo monitorizar la operación del dispositivo utilizando una conexión por puerto serie. El programa que ha sido usado para la lectura del puerto seria ha sido HyperTerminal (descargable desde la página web: http://www.hilgraeve.com/htpe/download.html). Para la supervisión del modulo a través del Puerto serie, debemos entender cómo funciona en realidad. Se incluye documentación y ejemplos del protocolo usado por VRbot en el documento adjunto [9], que también puede descargarse desde la página web del proveedor.

VRbot se comunica a través del Puerto serie de manera bastante sencilla. Lee caracteres simples, que toma como instrucciones, y devuelve también caracteres simples como respuesta. A continuación se indican algunas de las salidas y entradas usadas por el módulo y que se requieren para la realización de pruebas.

Entradas (Introducidas en una sesión de consola de HyperTerminal)
'**b**': Instrucción que "despierta" al modulo de su estado original de bajo consume. Si el dispositivo no es despertado, no aceptará ninguna orden adicional.
'**i**': Preparar al dispositivo para el reconocimiento de ordenes predefinidas (SI).
'**d**': Preparar al dispositivo para el reconocimiento de órdenes enseñadas al mismo (SD).
'**A**','**B**'...'**Z**': Las letras mayúsculas son usadas por el modulo a modo de números siguiendo la siguiente regla A=0; B=1; C=2 etc.
' ': (Barra espaciadora) Para pedirle al módulo una respuesta referente a la instrucción previa.

Salidas (Leídas desde la sesión de consola de HyperTerminal)
'**w**': Dispositivo despierto.
'**o**': Instrucción aceptada.
'**v**': Instrucción no válida.
'**A**','**B**'...'**Z**': Números, de manera análoga a las entradas.
'**r**': Instrucción reconocida satisfactoriamente.
'**e**': Error. El dispositivo detecta una instrucción pero no puede identificarla con exactitud.
'**t**': Tiempo de espera agotado. Señal de salida si para el tiempo de espera definido (5 segundos por defecto) ninguna otra instrucción es identificada.

Ejemplo de algoritmo de prueba

Apertura y configuración de HyperTerminal
1- Abrir el programa HyperTerminal
2- Configurar los parámetros de la conexión del puerto serie tal que coincidan con los indicados en la sección 5.2.3.
3- Conectar el puerto COM adecuado (por defecto, COM 1) y abrir la consola.

Despertando al dispositivo
4- Despertar el sistema. Para conseguir esto, introducir la letra 'b' hasta que la salida del módulo sea 'o', indicando que está preparado.
5- Si el modulo no ha sido previamente con VRbot GUI, la primera respuesta que debería proporcionar es 'w', indicando que se ha despertado, y posteriormente 'o'. Si el módulo ya estaba activo (Ej. Ya se había usado con VRbot GUI) su respuesta será directamente 'o'. waken up.

Probando una instrucción SD (Ejemplo: Instrucción GO_HOME)

6- Presionar 'd'. No habrá respuesta, el dispositivo se prepara para el reconocimiento de órdenes SD.
7- Presionar 'B', número del Wordset en que la instrucción a reconocer está localizada. (Como ya se ha explicado, los números son 'A'=0, B='1', C='2'... Puesto que la instrucción GO_HOME está en el Wordset 1, presionamos 'B')
8- Pronunciar la instrucción antes de que el tiempo de espera se sobrepase (5 segundos).
9- Esperar a la respuesta del sistema.
10- Existen 3 posibles respuestas:

't', Tiempo de espera agotado.

'e', Error. Error en el reconocimiento o resultados ambiguos.

'r', Instrucción reconocida, las palabras se corresponden con los patrones definidos para alguna de las instrucciones definidas en el Wordset 1.

11- Si la instrucción reconocida es 't' o 'e', volver al paso 6. Si la respuesta obtenida fue 'r', presionar la barra espaciadora.
12- El dispositivo mostrará el número correspondiente a la instrucción reconocida para el Wordset definido. Si reconoce adecuadamente la instrucción GO_HOME, la respuesta debiera ser 'C', u orden 2 del Wordset 1.

Las instrucciones ya incluidas en el dispositivo (SI), pueden probarse presionando 'i' en lugar de 'd' en el paso 6. El lenguaje por defecto para estas instrucciones es el inglés

El modulo puede ser configurado, enseñado y probado siguiendo este procedimiento.

5.2.6.- Programando VRbot utilizando un PC

Hasta ahora, ya hemos podido comprobar cómo funciona VRbot utilizando el puerto serie. El siguiente paso es el de crear un programa en C/C++ que nos permita probar el conjunto de VRbot + aplicación.

La mayor traba para conseguir este objetivo es la programación del puerto serie en si. Para lograrlo, se descargó código fuente de licencia GPL con las librerías y funciones pertinentes para poder programar tanto en entornos Linux como Windows (con el mismo código fuente). Este código pudo encontrarse en la página web http://www.teuniz.net/RS-232/ [10].

Se requieren 2 ficheros en este proyecto para conectar, programar y probar VRbot utilizando el puesto serie con un PC: *rs232.c* y *rs232.h*. El fichero de cabecera *rs232.h* contiene todas las definiciones necesarias usadas por el código fuente *rs232.c*

Las funciones usadas del código contenido en *rs232.c* son las siguientes:

```
int OpenComport(int comport_number, int baudrate)
```
Abre el puerto COM número “comport_number”. “baudrate” está expresado en baudios por segundo (Ej. 115200). Devuelve “1” en caso de error.

```
int PollComport(int comport_number, unsigned char *buf, int size)
```
Toma caracteres del puerto serie “comport_number” (si es que existen). “buf” es un puntero al buffer y “size” el tamaño del buffer en bytes.

```
int SendByte(int comport_number, unsigned char byte)
```
Envío de un byte a través del puerto serie “comport_number”. Devuelve “1” en caso de error.

```
void CloseComport(int comport_number)
```
Cierra el puerto serie “comport_number”.

Con el uso de estas 4 funciones podemos realizar nuestra programación a través del puerto serie.

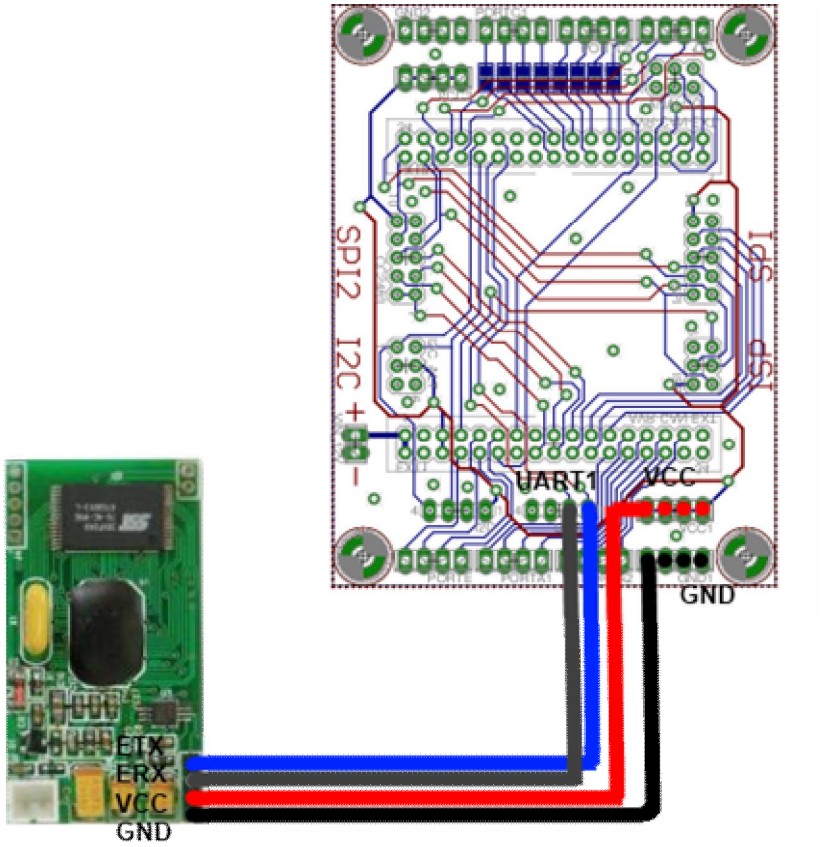
El programa creado es una simulación de la operación normal del soporte: El modulo tratará constantemente de reconocer la palabra de disparo con un tiempo de espera de 5 segundos. Si la palabra es reconocida, esperará entonces al reconocimiento de alguna de las instrucciones que tiene almacenadas. El dispositivo imprimirá por pantalla todas sus respuestas, preguntando por palabras de disparo, órdenes reconocidas y errores, pero de una manera más agradable que la definida por defecto, no únicamente con caracteres simples.

Este ejemplo de programación se incluye en el apéndice D, y como anexo [11].

5.2.7.- Conexión de VRbot y configuración del sistema conjunto

Hemos conseguido programa a través del Puerto serie y el modulo funciona, por lo que es momento de conectar VRbot al soporte robótico para añadir la capacidad de reconocimiento de voz.

Puesto que la UART0 está siendo usada para la programación, depuración y supervisión del sistema, el módulo VRbot estará conectado a la UART1 del AT90CAN128 tal u como se muestra en la figura (5.7)



(5.7) Diagrama de conexiones VRbot-PCB

Una vez se han hecho todas las conexiones, debe prepararse al microcontrolador para leer la UART1 en cualquier momento, por lo que ésta debe ser configurada de manera análoga a la UART0. Esto puede hacerse fácilmente desde el código fuente de *main.c*.

Es recomendable exportar el código ya desarrollado en la sección 5.2.6 y probarlo en el entorno del microcontrolador. Por supuesto, todas las librerías y funciones previamente usadas en los ficheros *rs232.c* and *rs232.h* deberán sustituirse por aquellas que configuran el puerto serie del microcontrolador, esto es, los ficheros *uart.c*, *uart.h*, *uart2.c* y *uart2.h*.

5.2.8.-Implementación del reconocimiento de voz en el sistema completo

En la anterior sección logramos que VRbot funcionara satisfactoriamente en conjunción con el microcontrolador del soporte robótico, por lo que finalmente podemos añadir la capacidad del reconocimiento de voz al sistema. Para establecer una separación lógica entre los modos manual y automático, el reconocimiento de voz solo se implementará en el modo automático, y trabajará en conjunción con el mando de control en este modo.

El código es similar al usado en el ejemplo para PC mostrado en la sección 5.2.6, y el algoritmo usado será el siguiente:

1) Inicialización del modulo de VRbot cuando el sistema principal es encendido.
2) En la rutina principal, proporcionar las órdenes adecuadas para que el módulo trate de reconocer la palabra de disparo continuamente. Mientras hace esto, el modo automático funciona de la manera habitual mediante el uso del mando de control.
3) Si el sistema se encuentra en modo automático y la palabra de disparo es reconocida, el módulo esperará entonces una instrucción válida. Si la instrucción no es reconocida, volver al punto 2.
4) Si la orden es identificada, la acción pedida se lleva a cabo. Independientemente de la orden reconocida, volver al paso 2.

5.2.9.- Funcionamiento del sistema final

El algoritmo implementado previamente en 5.2.8 es funcional. No obstante, deben realizarse cambios en el sistema de reconocimiento de voz para que su funcionamiento sea más seguro. Al realizar los primeros ensayos sobre el sistema completo, salió a la luz un defecto funcional del módulo de reconocimiento de voz: Los patrones para las palabras de disparo eran demasiado permisivos. Esto genera un problema porque provoca un gran número de falsos reconocimientos para estas palabras. Esto es, el usuario pronuncia la palabra “función”, y la palabra de disparo “instrucción” es reconocida por el sistema. Esta problemática no puede ser solucionada mediante un cambio en el código ni en la configuración del módulo. Al contrario que para los otros grupos de órdenes, no es posible cambiar la permisividad de los patrones de reconocimiento para las palabras de disparo.

La solución a este problema fue la de enseñar la palabra “instrucción” al modulo y asignarla a uno de los grupos de palabras de órdenes, pero en un grupo distinto al resto (en el grupo 4). De este modo, el sistema estará continuamente pidiendo órdenes del grupo 4, siendo su única palabra “instrucción”. En caso de que el reconocimiento sea exitoso, esperará entonces a que la siguiente palabra pertenezca al grupo 3 (Órdenes en Español). Puesto que los patrones reconocimiento para las instrucciones SD están configurados a “hard”, los falsos reconocimientos desaparecen.

6. Pruebas finales

Una vez hemos desarrollado todas las mejoras, debemos someter el sistema final a una serie de pruebas para determinar si este proyecto ha cumplido su propósito inicial de mejora del prototipo original. Todas y cada una de las funciones y arreglos realizados han sido testeadas exhaustivamente.

6.1.- Probando los modos manual y automático

Todas las mejoras implementadas en ambos modos funcionan correctamente. En el modo manual, el control manual de la tensión, el control asistido y la detección de colisiones han sido implementadas satisfactoriamente.

En el modo automático, se han añadido las funcionalidades de tensar/destensar el soporte y la detección de colisiones. Además, los movimientos ahora pueden ser detenidos si el usuario lo requiere, por lo que no es necesario que el soporte finalice una instrucción para que pueda recibir la siguiente orden.

Uno de los errores del prototipo original no ha podido ser resuelto. El movimiento durante la tensión/destensión sigue siendo asimétrico. La barra se extiende y queda firme correctamente, pero al contraerse, no recupera su posición original. Debe usarse el modo manual para contraer la barra hasta su posición original. Este problema ha sido estudiado a fondo pero no puede solucionarse únicamente con un cambio en el código, pues el error viene dado por un funcionamiento erróneo del motor. El motor ha sido testado con una fuente DC y monitorizado mediante el uso de polímetros y osciloscopios y no funciona correctamente. Cuando el soporte está destensándose, no lo hace completamente, incluso aunque el motor reciba la señal apropiada. La mayor parte de las veces se necesitan varios pulsos en la señal para conseguirlo, es decir, destensar un poco, parar, destensar otro poco, pasar, etc.

Es también importante remarcar que los valores umbral definidos para el acelerómetro son bastante estrictos. Una inclinación leve de la barra puede prevenir al sistema de llevar a cabo cualquier orden de movimiento. Aunque por regla general este es un efecto deseado, a veces provoca que el soporte se detenga en la mitad de su trayectoria, como por ejemplo si está moviéndose desde “Home” a “Chair”, lo cual es un efecto negativo en el modo manual.

6.2.- Probando el reconocimiento de voz

En este proyecto se ha implementado reconocimiento de órdenes por voz. Se hace necesario, por tanto, determinar su funcionalidad. Tras realizar las pruebas en el rango de 2 metros desde el micrófono, pueden extraerse las siguientes conclusiones:

- El reconocimiento es preciso, no se producen “falsos reconocimientos”. Esto es especialmente importante para la palabra de disparo, para que no sea reconocida en una conversación normal.
- Tasa de reconocimiento aproximada del 90% cuando el soporte está estático. Los patrones de reconocimiento son bastante estrictos, por lo que las instrucciones no son siempre reconocidas.

- La tasa de reconocimiento disminuye cuando el soporte se está moviendo, debido al ruido generado por los dos motores durante su funcionamiento y la cercanía del micrófono a los mismos. Este problema puede ser solventado parcialmente si se habla más alto.
- La tasa de reconocimiento disminuye de manera drástica si el usuario que da las órdenes es diferente al usuario que enseñó al módulo. Las instrucciones SD (Speaker Dependent) son realmente eso, dependientes del interlocutor.
- El reconocimiento de voz tiene lugar en el modo automático, pero durante el modo manual se siguen reconociendo palabras. Habitualmente esto no es un problema, pero si una palabra es reconocida en el modo manual, y el usuario cambia al modo automático, la instrucción reconocida se llevará a cabo.
- La tasa de reconocimiento mejora si el lenguaje usado es Español o Finés, puesto que son idiomas más fáciles de pronunciar y con menor propensión a ser afectados por el idioma original del usuario. No puede ignorarse que el modulo fue enseñado y entrenado por una persona española, por lo que el acento del inglés hablado difiere del inglés hablado por un finés (como por ejemplo todos aquellos que me ayudaron a realizar pruebas sobre el sistema).

6.3.- Problemas de inicialización

Durante todo el proyecto se ha detectado un error persistente que ya se encontraba en el prototipo original. La inicialización del microcontrolador no siempre se hace de manera correcta. Hay dos principales problemas que suceden a veces durante la inicialización:

- Tras la carga del código al microcontrolador, el sistema comienza a funcionar automáticamente, pero los valores del eje “Y” del acelerómetro difieren de los valores habituales de funcionamiento. Este error sucede cada vez que se carga código nuevo, y parece estar relacionado con el protocolo de comunicación I2C, o bien con el driver del acelerómetro LIS3LV02 usado (ya que este driver ha sido modificado). Este error se soluciona automáticamente si el sistema es apagado y encendido de nuevo.

- El programa no siempre ejecuta el bucle `while(1)` en la función `main()` al principio. Es un error extraño, pues en `main()`, los motores, así como la máquina de estados son inicializados. El sistema, entonces, comienza su operación normal, pero puesto que la rutina correspondiente al reconocimiento de voz tiene lugar dentro del bucle `while(1)` dentro de `main()`, no llega a funcionar. Es aún más extraño cuando durante la operación normal, el sistema entra de repente en el bucle y el reconocimiento de voz empieza. Esto sucede de manera aleatoria, a veces comenzando a los pocos segundos, mientras que otras veces tarda entre uno y dos minutos. Se ha probado a usar el sistema sin interrupciones (y por tanto sin máquina de estados) y con interrupciones a diferentes frecuencias para determinar si el error se debe a la sobresaturación del microcontrolador, pero el problema sigue sucediendo. Este error no sucede si el código se acaba de cargar y el sistema no se reinicia, pero esto crea un conflicto con el error del acelerómetro previamente comentado.

7. Conclusiones y desarrollos futuros

Conclusiones

Tal y como muestran las pruebas finales, se han realizado numerosas mejoras sobre el sistema original. No obstante, mejoras adicionales serían necesarias para que el prototipo funcionara correctamente en esta etapa de desarrollo:

- Reemplazo del motor de tensado/destensado:

Este motor no está funcionando adecuadamente y no es seguro para el soporte robótico el seguir usándolo. El problema no puede ser solucionado completamente mediante software. Además, este motor es relativamente ruidoso, creando un conflicto con el reconocimiento de voz, además de no ser agradable para el usuario. El problema del ruido está también presente en el otro motor.

- Respuesta sonora en el reconocimiento de voz:

Actualmente se necesita un PC para monitorizar la operación del reconocimiento de voz. El usuario desconoce si la palabra de disparo, o cualquier otra instrucción, ha sido reconocida a menos que un PC no esté siendo usado (a menos que ambas palabas sean reconocidas y el resultado visible). Esto no es aceptable fuera del entorno de desarrollo y pruebas.

- Considerar un sistema de reconocimiento de voz mejorado, o integrar una función para poder enseñar el reconocimiento de voz desde el propio soporte. El dispositivo tiene una gran tasa de error reconociendo las palabras de otros usuarios. Este problema solo es visible en las instrucciones SD, y no en las SI, por lo que es probable que en el futuro, tras una actualización de firmware, el comportamiento del módulo mejore. Si esto no sucede, sería recomendable el uso de un módulo más evolucionado.

- Corrección de los errores de inicialización comentados en la sección 6.3. No son aceptables en un sistema comercial.

Desarrollos futuros

El prototipo sigue estando en una etapa prematura de desarrollo. Todavía se requiere una gran cantidad de trabajo en mejoras adicionales, así como en mejorar las ya implementadas. En este momento, la utilidad del soporte robótico es limitada, y ciertamente se necesita una gran labor de pulido, tanto estética como funcional, antes de que pueda salir de la etapa de desarrollo y llegar al mercado. No obstante, el claro que una evolución del soporte robótico de apoyo sería bienvenido en hospitales, centros de asistencia y hogares privados.

Se espera que el campo de la Robótica Asistida Social alcance el mercado masivo en el futuro. Es importante, por tanto, el desarrollo continuo de nuevos robots y dispositivos para poder competir en el mercado. La versatilidad, aceptación por parte de los usuarios y los costes de producción de estos productos serán la clave para futuros desarrollos.

8. Documentación adjunta en el DVD

[1] Hoja de características del Módulo de Control Inteligente PIM3605

[2] Guía de usuario del PIM3605

[3] Hoja de características del microcontrolador AT90CAN128

[4] Hoja de características del acelerómetro LIS3L02

[5] Código fuente del prototipo original

[6] Cargando el código en el microcontrolador: Bootloader (*)

[7] Hoja de características de VRbot

[8] Guía de usuario de VRbot

[9] Protocolo serie de VRbot (*)

[10] Librerías para la programación del puerto serie

[11] Ejemplo de programación del puerto serie (*)

[12] Código fuente final (usado en los test finales)

[13] Código fuente final comentado (*)

[14] Presentación en video del soporte robótico

[15] VRbot GUI 1.1.3

[16] HyperTerminal Private Edition 6.3

(*) También en los apéndices

9. Referencias

(1) "The Engineering Handbook of Smart Technology for Aging, Disability and Independence"; Helal, Mokhtari and Abdulrazak; 2008.

(2) "Defining Socially Assistive Robotics". David Feil-Seifer and Maja J. Mataric. Interaction Laboratory. University of Southern California. In proceedings of the 2005 IEEE 9th International Conference on Rehabilitation Robotics June 28 – July 1, 2005, Chicago, IL, USA. http://cres.usc.edu/pubdb_html/files_upload/442.pdf (09/01/2010)

(3) Página web de Technosoft http://www.technosoftmotion.com/index.php (09/01/2010)

(4) Información, hoja de características y guía de usuario del PIM3605 http://www.technosoftmotion.com/products/OEM_PROD_PIM3605.htm (09/01/2010)

(5) Página web de Atmel http://www.atmel.com/ (09/01/2010)

(6) Información y hoja de características del AT90CAN128 http://www.atmel.com/dyn/Products/product_card.asp?part_id=3388 (09/01/2010)

(7) Hoja de características del LIS3L02 http://www.chipdocs.com/datasheets/datasheet-pdf/SGSThomson-Microelectronics/LIS3L02.html (09/01/2010)

(8) Proyecto Ceilbot http://autsys.tkk.fi/en/Ceilbot (09/01/2010)

(9) Página web de VeeaR http://vee-ar.com/ (09/01/2010)

(10) Información y hoja de características de VRbot http://www.veear.eu/Products/VRbot.aspx (09/01/2010)

(11) VRbot GUI (Descarga) http://www.veear.eu/LinkClick.aspx?fileticket=KZULbeOqvCM%3d&tabid=220&mid=626 (09/01/2010)

(12) Guía de usuario de VRbot (Descarga) http://www.veear.eu/LinkClick.aspx?fileticket=VowGK1u9rRc%3d&tabid=220&mid=620 (09/01/2010)

(13) Protocolo de comunicación de VRbot (Descarga) http://www.veear.eu/LinkClick.aspx?fileticket=gJhfDlDWpC8%3d&tabid=220&mid=626 (09/01/2010)

(14) Protocolo de comunicación en serie
http://en.wikipedia.org/wiki/Serial_port (09/01/2010)

(15) HyperTerminal
http://www.hilgraeve.com/htpe/download.html (09/01/2010)

(16) Librerías RS-232 C/C++
http://www.teuniz.net/RS-232/ (09/01/2010)

(17) AVRdude
http://www.bsdhome.com/avrdude/ (09/01/2010)

(18) Eclipse C/C++ IDE
http://www.eclipse.org/ (09/01/2010)

(19) AVR Eclipse plug-in
http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_Eclipse_Plugin
(09/01/2010)

(20) Embarcadero RAD Studio 2010
https://downloads.embarcadero.com/free/rad_studio (09/01/2010)

(21) “A book on C”; Kelley and Pohl; 1990.

Apéndice A

Carga del código en el microcontrolador: Software necesario y Bootloader

Este apéndice es un pequeño tutorial sobre cómo descargar el código escrito en el microcontrolador utilizando una conexión en serie.

A.1) Software necesario

Para poder programar el microcontrolador, es necesario obtener ciertos paquetes de software primero, listados a continuación:

- Linux OS
- Eclipse C/C++ IDE
- AVR Eclipse Plug-In
- AVRdude

Linux/Unix OS

Este proyecto se ha desarrollado utilizando el sistema operativo Linux Ubuntu 9.04, pero cualquier distribución basada en Debian debería servir también. Probablemente funcione bajo otros sistemas operativos, pero no han sido probados.

AVRdude

AVRdude es un paquete de software GPL que permite a los ordenadores funcionando bajo sistemas operativos Linux/Windows la carga/descarga de código a diversos dispositivos electrónicos. Para poder instalarlo (bajo Linux Ubuntu), debe introducirse el siguiente código en la consola (se requieren permisos de administrador):

```
user@user-pc:~$ sudo apt-get install avrdude
```

Eclipse C/C++ IDE

Eclipse C/C++ IDE es una ponderosa herramienta de desarrollo GPL para programadores, capaz de gestionar y depurar proyectos de gran escala con facilidad. El programa puede descargarse desde su página oficial:
http://www.eclipse.org/cdt/

AVR Eclipse Plug-In

Para que Eclipse reconozca las librerías e instrucciones usadas por AVR, es necesario descargar el AVR Eclipse plug-in. Instrucciones detalladas sobre como conseguir el plug-in pueden encontrarse en la página:
http://avr-eclipse.sourceforge.net/wiki/index.php/The_AVR_Eclipse_Plugin

A.2) Bootloader

El microcontrolador y su PCB tienen un “Bootloader” integrado. Con este “Bootloader”, y junto con las herramientas de software listadas previamente, cualquier código podrá ser descargado en el

microcontrolador. Para ello, deben seguirse los siguientes pasos:

Compilar el programa con “make”

Pese a que el programa ha sido escrito con Eclipse, para compilarlo debemos utilizar la consola de Linux. Debe escribirse un complejo fichero makefile y tras esto, compilar el programa ejecutando el comando “make” en la consola. Cuando se ejecute, se obtendrá información acerca de la memoria usada en el dispositivo. Ver a continuación:

```
user@user-pc:~/climbsupportpole$ make

... (Información acerca de la compilación, errores, avisos, etc.)

AVR Memory Usage
----------------
Device: at90can128

Program:   17236 bytes (13.2% Full)
(.text + .data + .bootloader)

Data:        397 bytes (9.7% Full)
(.data + .bss + .noinit)
```

El fichero makefile de este proyecto se incluye en el Apéndice B. Para entender cómo ha de ser escrito este fichero preparado para su uso con AVRdude, es recomendable la lectura de la guía de usuario de AVRdude:
http://www.nongnu.org/avrdude/user-manual/avrdude.html

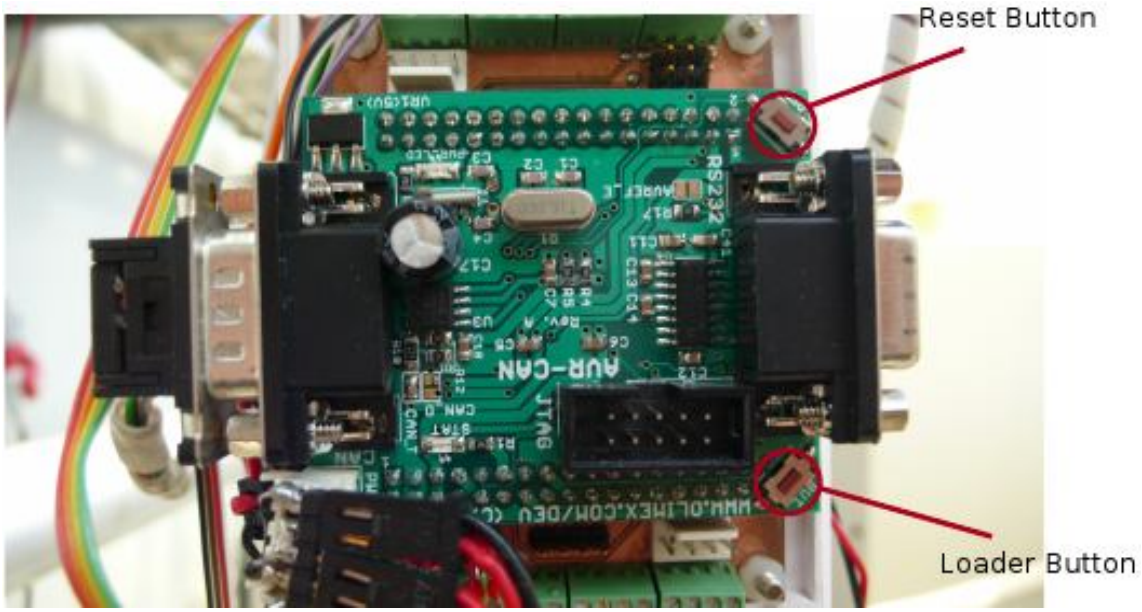
Poner el microcontrolador en modo “Bootloader”

Para cargar el código al microcontrolador, es necesario que éste este en modo “Bootloader”. En la PCB del microcontrolador pueden verse dos botones usados para este propósito: “Reset” y “Loader”. Ver figura (A.1)

Para poner el microcontrolador en modo “Bootloader” han de seguirse los siguientes pasos en un orden estricto:

1) Presionar (y mantener presionado) el botón “Reset”
2) Presionar (y mantener presionado) el botón “Loader”
3) Soltar el botón “Reset”
4) Soltar el botón “Loader”

Tras hacer esto, el microcontrolador entrará en modo “Bootloader” y el código podrá ser cargado.



(A.1) Localización del los botones “Reset” y “Loader” en la PCB del microcontrolador

Cargando el código

Cuando el dispositivo se halla en el modo “Bootloader”, para cargar el código es necesario conectar la PCB al PC usando un conector en serie (la PCB tiene un conector DB9 para ello) e introducir el siguiente comando en la consola (este código se incluye también en el fichero makefile):

```
user@user-pc:~$ avrdude -c avr911 -P /dev/ttyUSB1 -b 115200 -p at90can128 -U
flash:w:main.hex:i
```

AVRdude debería mostrar información relativa al programa a cargar:

```
Connecting to programmer: .
Found programmer: Id = "AVRBOOT"; type = S
… … … …
… … … … (Información sobre el proceso de descarga)
… … … …
avrdude: verifying ...
avrdude: 17108 bytes of flash verified

avrdude done.  Thank you.
```

Reseteando el microcontrolador

Tras la carga del código, el sistema comenzará su funcionamiento, pero hasta que los errores de inicialización listados en la sección 6.3 sean arreglados, es necesario apagar el dispositivo y reiniciarlo, no sólo resetearlo presionando el boto “Reset” en la PCB.

Monitorizando el soporte robótico

Puede monitorizarse el funcionamiento del programa a través de una consola que nos muestre el puerto serie. Para ello, introducir el siguiente comando en la consola:

```
user@user-pc:~$screen /dev/ttyUSB1

HELLO!
Initializing device
INIT DRIVE
INIT DRIVE OK
NOT TIGHTENED
GOING HOME...
AT HOME
AUTO
```

Apédice B

Makefile del programa

Este fichero makefile fue creado originalmente por Teemu Kuusisto, siendo modificado en este proyecto para que se adecúe a la adición de nuevos ficheros, así como el renombramiento de otros tantos.

```
#############################################################################
# Makefile for the support pole project
#############################################################################

## General Flags
PROJECT = main
MCU = at90can128
TARGET = main.elf
CC = avr-gcc
#avr-gcc.exe

FCPU=16000000
## Options common to compile, link and assembly rules
COMMON = -mmcu=$(MCU) -DF_CPU=$(FCPU)UL

## Compile options common for all C compilation units.
CFLAGS = $(COMMON)
CFLAGS += -Wall -gdwarf-2 -std=gnu99 -Os -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums
CFLAGS += -MD -MP -MT $(*F).o -MF dep/$(@F).d

## Assembly specific flags
ASMFLAGS = $(COMMON)
ASMFLAGS += $(CFLAGS)
ASMFLAGS += -x assembler-with-cpp -Wa,-gdwarf2

## Linker flags
LDFLAGS = $(COMMON)
LDFLAGS +=  -Wl,-Map=main.map


## Intel Hex file production flags
HEX_FLASH_FLAGS = -R .eeprom

HEX_EEPROM_FLAGS = -j .eeprom
HEX_EEPROM_FLAGS += --set-section-flags=.eeprom="alloc,load"
HEX_EEPROM_FLAGS += --change-section-lma .eeprom=0 --no-change-warnings


## Objects that must be built in order to link
OBJECTS = main.o statemachine.o leds.o interrupts.o motor.o slidepotentiometer.o accelerometer.o
lis3lv02_driver.o a2d.o i2c.o uart2.o rprintf.o buffer.o can_drv.o can_lib.o canproto.o voicerecon.o

## Objects explicitly added by the user
LINKONLYOBJECTS =
```

```
## Build
all: $(TARGET) main.hex main.eep main.lss size

flash: main.hex
	avrdude -c avr911 -P COM1 -b 115200 -p at90can128 -U flash:w:main.hex:i

flashlinux:
	avrdude -c avr911 -P /dev/ttyUSB1 -b 115200 -p at90can128 -U flash:w:main.hex:i


## Compile
main.o: main.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

statemachine.o: statemachine.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

leds.o: leds.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

interrupts.o: interrupts.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

motor.o: motor.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

slidepotentiometer.o: slidepotentiometer.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

accelerometer.o: accelerometer.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

lis3lv02_driver.o: lis3lv02_driver.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

a2d.o: a2d.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

i2c.o: i2c.c
	$(CC) $(INCLUDES) $(CFLAGS) -c -fgnu89-inline $<

uart2.o: uart2.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

rprintf.o: rprintf.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

buffer.o: buffer.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

canproto.o: canproto.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

can_lib.o: can_lib.c
```

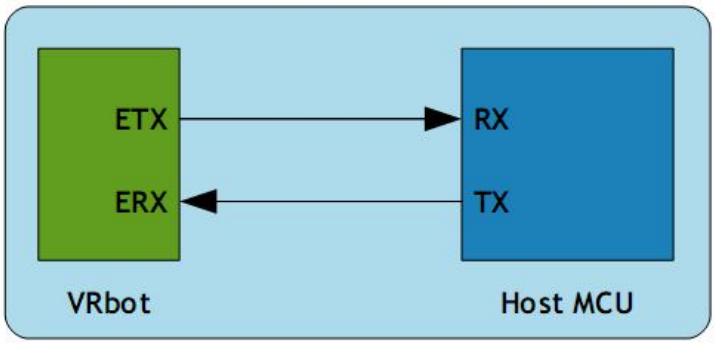
```
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

can_drv.o: can_drv.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

voicerecon.o: voicerecon.c
	$(CC) $(INCLUDES) $(CFLAGS) -c  $<

##Link
$(TARGET): $(OBJECTS)
	 $(CC) $(LDFLAGS) $(OBJECTS) $(LINKONLYOBJECTS) $(LIBDIRS) $(LIBS) -o $(TARGET)

%.hex: $(TARGET)
	avr-objcopy -O ihex $(HEX_FLASH_FLAGS)  $< $@

%.eep: $(TARGET)
	-avr-objcopy $(HEX_EEPROM_FLAGS) -O ihex $< $@ || exit 0

%.lss: $(TARGET)
	avr-objdump -h -S $< > $@

size: ${TARGET}
	@echo
	@avr-size -C --mcu=${MCU} ${TARGET}

## Clean target
.PHONY: clean
clean:
	-rm -rf $(OBJECTS) main.elf dep/* main.hex main.eep main.lss main.map


## Other dependencies
-include $(shell mkdir dep 2>/dev/null) $(wildcard dep/*)
```

Appendix B
VRbot Communication Protocol

Protocol and Interface Basics

Communication with the VRbot module uses a standard UART interface compatible with 3.3-5V TTL logical levels, according to the powering voltage VCC.

A typical connection to an MCU-based host:



The initial configuration at power on is 9600 baud, 8 bit data, No parity, 1 bit stop. The baud rate can be changed later to operate in the range 9600 - 115200 baud.

The communication protocol only uses printable ASCII characters, which can be divided in two main groups:

- Command and status characters, respectively on the TX and RX lines, chosen among lower-case letters
- Command arguments or status details, again on the TX and RX lines, spanning the range of capital letters

Each command sent on the TX line, with zero or more additional argument bytes, receives an answer on the RX line in the form of a status byte followed by zero or more arguments.

There is a minimum delay before each byte sent out from the VRbot module to the RX line, that is initially set to 20 ms and can be selected later in the ranges 0 - 9 ms, 10 - 90 ms, 100 ms - 1 s. That accounts for slower or faster host systems and therefore suitable also for software-based serial communication (bit-banging).

The communication is host-driven and each byte of the reply to a command has to be acknowledged by the host to receive additional status data, using the *space* character. The reply is aborted if any other character is received and so there is no need to read all the bytes of a reply if not required.

Invalid combinations of commands or arguments are signaled by a specific status byte, that the host should be prepared to receive if the communication fails. Also a reasonable timeout should be used to recover from unexpected failures.

If the host does not send all the required arguments of a command, the command is ignored by the module, without further notification, and the host can start sending another command.

The module automatically goes to lowest power sleep mode after power on. To initiate communication, send any character to wake-up the module.

Command Details

Format of command strings accepted by the module. Please note that numeric arguments of command requests are mapped to upper-case letters (see the related section).

CMD_KNOB	
`'k'` (6Bh)	Set SI knob to specified level
`[1]`	Confidence threshold level (0-4): 0= loosest:more valid results 2= typical value (default) 4= tightest:fewer valid results NOTE: knob is ignored for trigger words
Expected replies: STS_SUCCESS	

CMD_LEVEL	
`'v'` (76h)	Set SD level
`[1]`	Strictness control setting (1-5): 1 = easy, 2 = default, 5 = hard A higher setting will result in more recognition errors.
Expected replies: STS_SUCCESS	

CMD_LANGUAGE	
`'l'` (6Ch)	Set SI language
`[1]`	Language (0 = English, 1 = Italian, 2 = Japanese, 3 = German)
Expected replies: STS_SUCCESS	

CMD_BREAK	
`'b'` (62h)	Abort recognition in progress if any or do nothing **Known issues:** In firmware ID 0, any other character received during recognition will prevent this command from stopping recognition, that will continue until timeout or other recognition results.
Expected replies: STS_SUCCESS, STS_INTERR	

CMD_SLEEP	
`'s'` (73h)	Go to the specified power-down mode
`[1]`	Sleep mode (0-8): 0 = wake on received character only 1 = wake on whistle or received character 2 = wake on loud sound or received character 3-5 = wake on double clap (with varying sensitivity) or received character 6-8 = wake on triple clap (with varying sensitivity) or received character
Expected replies: STS_SUCCESS	

CMD_TIMEOUT	
'o' (6Fh)	Set recognition timeout
[1]	Timeout (-1 = default, 0 = infinite, 1-31 = seconds)
Expected replies: STS_SUCCESS	

CMD_RECOG_SI	
'i' (69h)	Activate SI recognition from specified wordset
[1]	Wordset index (0-3)
Expected replies: STS_SIMILAR, STS_TIMEOUT, STS_ERROR	

CMD_TRAIN_SD	
't' (74h)	Train specified SD/SV command
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Command position (0-31)
Expected replies: STS_SUCCESS, STS_RESULT, STS_SIMILAR, STS_TIMEOUT, STS_ERROR	

CMD_GROUP_SD	
'g' (67h)	Insert new SD/SV command
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Position (0-31)
Expected replies: STS_SUCCESS, STS_OUT_OF_MEM	

CMD_UNGROUP_SD	
'u' (75h)	Remove SD/SV command
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Position (0-31)
Expected replies: STS_SUCCESS	

CMD_RECOG_SD	
'd' (64h)	Activate SD/SV recognition
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
Expected replies: STS_RESULT, STS_SIMILAR, STS_TIMEOUT, STS_ERROR	

CMD_ERASE_SD	
'e' (65h)	Erase training of SD/SV command
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Command position (0-31)
Expected replies: STS_SUCCESS	

CMD_NAME_SD	
'n' (6Eh)	Label SD/SV command
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Command position (0-31)
[3]	Length of label (0-31)
[4-n]	Text for label (ASCII characters from 'A' to '`')
Expected replies: STS_SUCCESS	

CMD_COUNT_SD	
'c' (63h)	Request count of SD/SV commands in the specified group
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
Expected replies: STS_COUNT	

CMD_DUMP_SD	
'p' (70h)	Read SD/SV command data (label and training)
[1]	Group index (0 = trigger, 1-15 = generic, 16 = password)
[2]	Command position (0-31)
Expected replies: STS_DATA	

CMD_MASK_SD	
'm' (6Dh)	Request bit-mask of non-empty groups
Expected replies: STS_MASK	

CMD_RESETALL	
'r' (72h)	Reset all commands and groups
'R' (52h)	Confirmation character
Expected replies: STS_SUCCESS	

CMD_ID	
`'x'` (78h)	Request firmware identification
Expected replies: STS_ID	

CMD_DELAY	
`'y'` (79h)	Set transmit delay
`[1]`	Time (0-10 = 0-10 ms, 11-19 = 20-100 ms, 20-28 = 200-1000 ms)
Expected replies: STS_SUCCESS	

CMD_BAUDRATE	
`'a'` (61h)	Set communication baud-rate
`[1]`	Speed mode (1 = 115200, 2 = 57600, 3 = 38400, 6 = 19200, 12 = 9600)
Expected replies: STS_SUCCESS	

Status Details

Replies to commands follow this format. Please note that numeric arguments of status replies are mapped to upper-case letters (see the related section).

STS_MASK	
`'k'` (6Bh)	Mask of non-empty groups
`[1-8]`	4-bit values that form 32-bit mask, LSB first
In reply to: CMD_MASK_SD	

STS_COUNT	
`'c'` (63h)	Count of commands
`[1]`	Integer (0-31)
In reply to: CMD_COUNT_SD	

STS_AWAKEN	
`'w'` (77h)	Wake-up (back from power-down mode)
In reply to: Any character after power on or sleep mode	

STS_SIMILAR	
`'s'` (73h)	Recognised SI word or Training similar to SI word
`[1]`	Word index (0-31)
In reply to: CMD_RECOG_SI, CMD_RECOG_SD, CMD_TRAIN_SD	

STS_OUT_OF_MEM	
`'m'` (6Dh)	Memory full error
In reply to: CMD_GROUP_SD	

STS_ID	
`'x'` (78h)	Provide firmware identification
`[1]`	Version identifier (0)
In reply to: CMD_ID	

STS_ERROR	
`'e'` (65h)	Signal recognition error
`[1-2]`	Two 4-bit values that form 8-bit error code (80h = NOTA, otherwise see FluentChip error codes)
In reply to: CMD_RECOG_SI, CMD_RECOG_SD, CMD_TRAIN_SD	

STS_INVALID	
`'v'` (76h)	Invalid command or argument
In reply to: Any invalid command or argument	

STS_TIMEOUT	
`'t'` (74h)	Timeout expired
In reply to: CMD_RECOG_SI, CMD_RECOG_SD, CMD_TRAIN_SD	

STS_INTERR	
`'i'` (69h)	Interrupted recognition
In reply to: CMD_BREAK while in training or recognition	

STS_DATA	
'd' (64h)	Provide command data
[1]	Training information (0-7 = training count, +8 = SD/SV conflict, +16 = SI conflict)
[2]	Conflicting command position (0-31)
[3]	Length of label (0-31)
[4-n]	Text of label (ASCII characters from 'A' to '`')
In reply to: CMD_DUMP_SD	

STS_SUCCESS	
'o' (6Fh)	OK or no errors status
In reply to: CMD_BREAK, CMD_DELAY, CMD_BAUDRATE, CMD_TIMEOUT, CMD_KNOB, CMD_LEVEL, CMD_LANGUAGE, CMD_SLEEP, CMD_GROUP_SD, CMD_UNGROUP_SD, CMD_ERASE_SD, CMD_NAME_SD, CMD_RESETALL	

STS_RESULT	
'r' (72h)	Recognised SD/SV command or Training similar to SD/SV command
[1]	Command position (0-31)
In reply to: CMD_RECOG_SD, CMD_TRAIN_SD	

Arguments Mapping

These are the characters used to represent integer values in the range -1 to 31 for command or status arguments.

ARG_MIN	
'@' (40h)	Minimum argument value (-1)

ARG_MAX	
'`' (60h)	Maximum argument value (+31)

ARG_ZERO	
'A' (41h)	Zero argument value (0)

ARG_ACK	
' ' (20h)	Read more status arguments

Communication Examples

These are some examples of actual command and status strings exchanged with the VRbot module by host programs and the expected program flow with pseudo-code sequences.

The pseudo-instruction `SEND` transmits the specified character to the module, while `RECEIVE` waits for a reply character (a timeout is not explicitly handled for simple commands, but should be always implemented if possible).

Also, the `OK` and `ERROR` routines are not explicitly defined, since they are host and programming language dependent, but appropriate code should be written to handle both conditions.

Lines beginning with a # (sharp) character are comments.

Please note that in a real programming language it would be best to define some constants for the command and status characters, as well as for mapping numeric arguments, that would be used throughout the program, to minimize the chance of repetition errors and clarify the meaning of the code.

See the header file *protocol.h* for sample definitions that can be used in a C language environment.

Here below all the characters sent and received are written explicitly in order to clarify the communication protocol detailed in the previous sections.

(1) Recommended wake up procedure:

```
# wake up or interrupt recognition or do nothing
# (use a timeout or max repetition count)
DO
     SEND 'b'
LOOP UNTIL RECEIVE = 'o'
```

(2) Recommended setup procedure:

```
# ask firmware id
SEND 'x'
IF NOT RECEIVE = 'x' THEN ERROR

# send ack and read status (expecting id=0)
SEND ' '
IF RECEIVE = 'A' THEN OK ELSE ERROR

# set language for SI recognition (Japanese)
SEND 'l'
SEND 'C'
IF RECEIVE = 'o' THEN OK ELSE ERROR

# set timeout (5 seconds)
SEND 'o'
SEND 'F'
IF RECEIVE = 'o' THEN OK ELSE ERROR
```

(3) Recognition of a built-in SI command:

```
# start recognition in wordset 1
SEND 'i'
SEND 'B'
# wait for reply:
# (if 5s timeout has been set, wait for max 6s then abort
#  otherwise trigger recognition could never end)
result = RECEIVE

IF result = 's' THEN
     # successful recognition, ack and read result
     SEND ' '
     command = RECEIVE - 'A'
     # perform actions according to command
ELSE IF result = 't' THEN
     # timed out, no word spoken
ELSE IF result = 'e' THEN
     # error code, ack and read which one
     SEND ' '
     error = (RECEIVE - 'A') * 16
     SEND ' '
     error = error + (RECEIVE - 'A')
     # perform actions according to error
ELSE
     # invalid request or reply
     ERROR
END IF
```

(4) Adding a new SD command:

```
# insert command 0 in group 3
SEND 'g'
SEND 'D'
SEND 'A'
IF RECEIVE = 'o' THEN OK ELSE ERROR

# set command label to "ARDUINO_2009"
SEND 'g'
SEND 'D'
SEND 'A'
SEND 'M'  # name length (12 characters)
SEND 'A'
SEND 'R'
SEND 'D'
SEND 'U'
SEND 'I'
SEND 'N'
SEND 'O'
SEND '_'
# encode each digit with a ^ prefix
# followed by the digit mapped to upper case letters
SEND '^'
SEND 'C'
SEND '^'
SEND 'A'
SEND '^'
SEND 'A'
SEND '^'
SEND 'J'
IF RECEIVE = 'o' THEN OK ELSE ERROR
```

(5) Training an SD command:

```
# repeat the whole training procedure twice for best results
# train command 0 in group 3
SEND 't'
SEND 'D'
SEND 'A'
# wait for reply:
#  (default timeout is 3s, wait for max 1s more then abort)
result = RECEIVE

IF RECEIVE = 'o' THEN
     # training successful
     OK
ELSE IF result = 'r' THEN
     # training saved, but spoken command is similar to
     # another SD command, read which one
     SEND ' '
     command = RECEIVE - 'A'
     # may notify user and erase training or keep it
ELSE IF result = 's' THEN
     # training saved, but spoken command is similar to
     # another SI command (always trigger, may skip reading)
     SEND ' '
     command = RECEIVE - 'A'
     # may notify user and erase training or keep it
ELSE IF result = 't' THEN
     # timed out, no word spoken or heard
ELSE IF result = 'e' THEN
     # error code, ack and read which one
     SEND ' '
     error = (RECEIVE - 'A') * 16
     SEND ' '
     error = error + (RECEIVE - 'A')
     # perform actions according to error
ELSE
     # invalid request or reply
     ERROR
END IF
```

(6) Read used command groups:

```
# request mask of groups in use
SEND 'm'
IF NOT RECEIVE = 'k' THEN ERROR
# read mask to 32 bits variable
# in 8 chunks of 4 bits each
SEND ' '
mask = (RECEIVE - 'A')
SEND ' '
mask = mask + (RECEIVE - 'A') * 2^4
SEND ' '
mask = mask + (RECEIVE - 'A') * 2^8
...
SEND ' '
mask = mask + (RECEIVE - 'A') * 2^24
```

(7) Read how many commands in a group:

```
# request command count of group 3
SEND 'c'
SEND 'D'
IF NOT RECEIVE = 'c' THEN ERROR
# ack and read count
SEND ' '
count = RECEIVE - 'A'
```

(8) Read a user defined command:

```
# dump command 0 in group 3
SEND 'p'
SEND 'D'
SEND 'A'
IF NOT RECEIVE = 'd' THEN ERROR
# read command data
SEND ' '
training = RECEIVE - 'A'
# extract training count (2 for a completely trained command)
tr_count = training AND 7
# extract flags for conflicts (SD or SI)
tr_flags = training AND 24
# read index of conflicting command (same group) if any
SEND ' '
conflict = RECEIVE - 'A'
# read label length
SEND ' '
length =  RECEIVE - 'A'
# read label text
FOR i = 0 TO length - 1
 SEND ' '
 label[i] = RECEIVE
 # decode digits
 IF label[i] = '^' THEN
 SEND ' '
 label[i] = RECEIVE - 'A' + '0'
 END IF
NEXT
```

Built-in Command Sets

In the tables below a list of all built-in commands for each supported language, along with group index (trigger or wordset), command index and language identifier to use with the communication protocol.

		Language			
		0	1	2	3
Trigger/Wordset	**Command Index**	**English (US)**	**Italian**	**Japanese**	**German**
0	0	robot	robot	ロボット	roboter
1	0	action	azione	アクション	aktion
	1	move	vai	ススメ	gehe
	2	turn	gira	マガレ	wende
	3	run	corri	ハシレ	lauf
	4	look	guarda	ミロ	schau
	5	attack	attacca	コーゲキ	attacke
	6	stop	fermo	トマレ	halt
	7	hello	ciao	こんにちわ	hallo
2	0	left	a sinistra	ヒダリ	nach_links
	1	right	a destra	ミギ	nach_rechts
	2	up	in alto	ウエ	hinauf
	3	down	in basso	シタ	hinunter
	4	forward	avanti	マエ	vorwärts
	5	backward	indietro	ウシロ	rückwärts
3	0	zero	zero	ゼロ	null
	1	one	uno	いち	eins
	2	two	due	ニ	zwei
	3	three	tre	サン	drei
	4	four	quattro	ヨン	vier
	5	five	cinque	ゴ	fünf
	6	six	sei	ロク	sechs
	7	seven	sette	ナナ	sieben
	8	eight	otto	ハち	acht
	9	nine	nove	クュー	neun
	10	ten	dieci	ジュー	zehn

Apéndice D

Ejemplo de programación del puerto serie en C/C++

Este código de ejemplo ha sido usado para realizar las pruebas de programación en serie a través de un PC del módulo de reconocimiento de voz VRbot. Hace uso de dos librerías de licencia GPL, rs-232.c y rs-232.h desarrolladas por Teunis van Beelen. Este programa fue compilado, ejecutado y probado con Embarcadero RAD Studio 2010 (Versión de prueba), sobre el sistema operativo Windows Vista.

Fichero de cabecera voicerecon.h

```
/* The following definitions help to work with the VRBot module environment
       Further details of the VRbot commands and responses can be found in
       the Appendix C of this project */

#define START_RECOGNITION 'd'
#define TRIGGER 'A'
#define INSTRUCTION 'B'
#define MOVE_LEFT 'A'
#define MOVE_RIGHT  'B'
#define STOP 'C'
#define TIGHTEN 'D'
#define GO_HOME 'E'
#define GO_TO_BED 'F'
#define GO_TO_CHAIR 'G'
#define CLOSE 'H'
#define ACK ' '
#define START_RECON 'd'
#define TRIGGER_RECON 'A'
#define ORDER_RECON 'B'
#define TIMEOUT 't'
#define RECON_OK 'r'
#define ASK_ID 'x'
#define ID0 'A'
#define WAKE 'b'
#define LANGUAGE_SET 'l'
#define ENGLISH 'A'
#define OK 'o'
#define FIVE_SECONDS 'F'
#define SET_TIMEOUT 'o'

/* Declaration of the functions used */
void initialization(unsigned char *buf);
void SendInstruction (unsigned char instruction, unsigned char *buf);
int GetInstruction(unsigned char *buf);
```

Fichero de código *voicerecon.c*

```
#include "VoiceRecon.h"
#include "rs-232.h"

/* This program has been created as a test of the VRbot module under a PC
       environment. It communicates with the device using the serial port.

       The program asks the user to say the trigger word 'instruction'.
       If the word is not recognized, will print the error, and if the user
       does not speak within 5 seconds, will print 'timeout'. If the word is
       recognized, it will ask for any of the commands. If any of the
       commands is recognized, it will print it, if not, it would be the same as
       for the trigger word. Either way it will start again asking for the
       trigger word. The only way of ending the program is to give it the
       command 'close'.In that case, it will close itself.
*/
int main (void)
{
       char info;
       int i, response;
       int close=0;
       unsigned char *buf;
       initialization(buf);
       printf("\nSTARTING RECOGNITION\n");

       while (close!=1){
              printf("Trigger Word: ");
              SendByte(0, START_RECON);
              SendInstruction(TRIGGER_RECON,buf);
              if (*buf==RECON_OK){
                     printf("ACCEPTED -- Instruction: ");
                     SendByte(0, START_RECON);
                     SendInstruction (ORDER_RECON,buf);
                     if (*buf==RECON_OK) {
                            SendInstruction (ACK,buf);
                            close=GetInstruction(buf);
                     } else if (*buf==TIMEOUT) {
                            printf("Timeout \n");
                     } else {
                            printf("Recognition failed\n");
                     }
              } else if (*buf==TIMEOUT) {
                     printf("Timeout \n");
              } else {
                     printf("Recognition failed\n");
              }
       }
       /* Closing COM port 0 to avoid conflicts with other programs
              that want to use it */
       printf("\nClosing port 0: ");
       CloseComport(0);
       printf("CLOSED\n");
       printf("Exiting program\n");
       return 0;
}


/* initialization(unsigned char *buf)
       *buf: Byte read from the serial port
```

```
        This function opens the COM port 0 to establish communication
        through the serial port and configures some features of the VRbot module*/

void initialization(unsigned char *buf)
{
        int response,i;
        i=0;
        /* Opening COM port 0, and establishing the parameters of the serial port
                communication as 8-bit data, 1-bit stop, no parity, 9600bd,
                no flow control */
        printf("Opening COM 0\n");
        OpenComport(0,9600);
        printf("Initializing device\n");
        // Waking up device from low-power/saving mode
        while (*buf != OK) {
                SendInstruction (WAKE,buf);
                if (i>=5) {
                        printf("ERROR: Failed to initialize device\n");
                        break;
                }
                i++;
        }
        // Ask firmware ID (It has to be '0')
        SendInstruction(ASK_ID,buf);
        if (*buf != ASK_ID) {
                printf("ERROR: Cannot read device ID\n");
        }
        // Send ack and read status (expecting ID=0)
        SendInstruction (ACK,buf);
        if (*buf != ID0) {
                printf("ERROR: Device ID doesn't match\n");
        }
        else {
                printf("Device identified\n");
        }
        // Set language for Speaker Independent instructions (English)
        SendByte(0,LANGUAGE_SET);
        SendInstruction (ENGLISH,buf);
        if (*buf != OK) {
                printf("ERROR: Could not set default language for built-in instructions (English)\n");
        } else {
                printf("Language set: English\n");
        }
        // Set timeout (5 seconds)
        SendByte(0,SET_TIMEOUT);
        SendInstruction (FIVE_SECONDS,buf);
        if (*buf != OK) {
                printf("ERROR: Timeout not set\n");
        }
        printf("Timeout set: 5 seconds\n");
        printf("Device initialized\n");
}


/* SendInstruction (unsigned char instruction, unsigned char *buf)
        instruction: Command (byte) sent to the device through the serial port
        *buf: Byte read from the serial port buffer

        This function sends an instruction to the device and then waits for
        a response from it  */
```

```
void SendInstruction (unsigned char instruction, unsigned char *buf)
{
	int response=0;
	SendByte(0, instruction);
	while (response!=1)
	{
		response=PollComport(0, buf, 1);
	}
}

/* GetInstruction(unsigned char *buf)
	*buf: Byte read from the serial port buffer

	This function gets the byte read from the device and translates the
	response to a more friendly language

	Always returns zero, save if the instruction 'close' is given
	In that case it will return one and the main routine will end */

int GetInstruction(unsigned char *buf)
{
	int close=0;
	switch (*buf) {
		case MOVE_LEFT:
			printf("MOVING LEFT...\n");
			break;

		case MOVE_RIGHT:
			printf("MOVING RIGHT...\n");
			break;

		case STOP:
			printf("STOP\n");
			break;

		case TIGHTEN:
			printf("TIGHTEN\n");
			break;

		case GO_HOME:
			printf("GOING HOME\n");
			break;

		case GO_TO_BED:
			printf("GOING TO BED\n");
			break;

		case GO_TO_CHAIR:
			printf("GOING TO CHAIR\n");
			break;

		case CLOSE:
			close=1;
			break;

	default:
		;
	}
	return close;
}
```

Apéndice E

Código implementado en el sistema final

En este apéndice se incluye el código usado en el sistema final. En la tabla a continuación se muestran los ficheros utilizados por el sistema, así como una breve descripción y el nombre de su/sus programadores. Para compilar el programa debe usarse el fichero makefile incluido en el apéndice B. Sólo aquellos ficheros desarrollados y escritos por estudiantes de la Universidad Alvar Aalto para este proyecto se han incluido en este documento. El resto de ficheros son genéricos y de propósito general, no desarrollados para este proyecto en particular, y pueden obtenerse de Atmel directamente o son GNU.

File name(*)	Functions / Devices managed	Programmer(**)
a2d.c /.h	Analog to digital conversion libraries	P.S.
accelerometer.c /.h	Accelerometer control routines	T.K. & J.L.
avrlibdefs.h	Definitions and macros for Atmel AVR series	P.S.
avrlibtypes.h	Type definitions for Atmel AVR series	P.S.
buffer.c /.h	Buffer libraries used by the UARTs	P.S.
buzzer.c /.h	Buzzer (not used)	T.K.
can_drv.c /.h	CAN drivers for Atmel AT90CAN series	Atmel
can_lib.c /.h	CAN function libraries for Atmel AT90CAN series	Atmel
canproto.c /.h	Modified CAN libraries for additional functions	J.A.
config.h	General system definitions	T.K.
global.h	Includes and definitions for Atmel AVR series	P.S.
i2c.c /.h	I2C protocol for Atmel AVR series	P.S.
i2cconf.h	I2C protocol for Atmel AVR series	P.S.
interrupts.c /.h	Interrupt for the state machine subroutine	T.K.
leds.c /.h	LEDs of the control pad	T.K.
lis3l02.c /.h	Accelerometer drivers for Atmel AVR series	P.S.
lis3l02_driver.c /.h	Modified and improved accelerometer drivers	A.L.
main.c	Main routine	T.K. & J.L.
motor.c /.h	Tightening/Loosening motor	T.K.
rprintf.c /.h	rprintf function for Atmel AVR series	P.S.
slidepotentiometer.c /.h	Sliding potentiometer	T.K.
statemachine.c /.h	State machine of the system program	T.K. & J.L.
timer.c /.h	System timer function libraries	P.S.
uart.c /.h	UART driver for Atmel AVR series	P.S.
uart2.c /.h	UART driver for Atmel AVR series	P.S.
util.h	Standard useful definitions	T.K.

(*) Filenames

Los ficheros usados se dividen en ficheros de código C (.c) y ficheros de cabecera (.h). La mayoría de los ficheros son de código C con sus respectivos ficheros de cabecera, de ahí la notación ".c /.h"

(**) Programadores
- **A.L.:** Antti Liesjärvi; Estudiante de la Universidad Alvar Aalto University, Departamento de Automática y Sistemas.
- Atmel: Ficheros de Atmel sobre el bus de comunicación CAN con copyright.
- J.A.: Johannes Aalto; Estudiante de la Universidad Alvar Aalto University, Departamento de Automática y Sistemas
- J.L.: Jorge Latorre; Estudiante de la Universidad Carlos III de Madrid; Estudiante de intercambio en la Universidad Alvar Aalto University, Departamento de Automática y Sistemas; Escritor de éste proyecto fin de carrera.
- T.K.: Teemu Kuusisto. Estudiante de la Universidad Alvar Aalto University, Departamento de Automática y Sistemas; Creador del prototipo original de soporte robótico.
- P.S.: Pascal Stang; Autor de los ficheros de Atmel bajo licencia GNU

Lista de ficheros incluidos en este proyecto

NOTA: Muchos de los comentarios incluidos en el código han sido incluidos **después** de que las pruebas finales hayan tenido lugar. Cualquier cambio que se haya podido hacer sobre los ficheros tras los últimos test es únicamente debido a la adición de comentarios. En caso de que aparezca algún error al compilar y probar este código, el último código probado se incluye en el DVD [12].

main.c

```
#define MAIN_C

/*
 * main() routine initializes all the I/O of the microcontroller, as well as
 * the devices attached to it. The voice recognition routine is executed here, but
 * the remaining control of the system is done by the interrupt driven
 * state_machine() function, in statemachine.c file
 */

#include <avr/io.h>
#include <avr/interrupt.h>
#include <util/delay.h>
#include "util.h"
#include "motor.h"
#include "statemachine.h"
#include "leds.h"
#include "interrupts.h"
#include "i2c.h"
#include "lis3lv02_driver.h"
#include "slidepotentiometer.h"
#include "a2d.h"
#include "accelerometer.h"
#include "uart2.h"
#include "rprintf.h"
#include "canproto.h"
#include "voicerecon.h"

void init(void);

unsigned char VRorder;
volatile uint8_t canstate;
volatile uint8_t state;
volatile uint8_t next_state;

//ID of the MICROController
#define CANID 2

int main(void) {
	// CAN initialization
	caninit();
	// Digital I/O initialization
	init();

	sei();

	while (1)
	{
		// Starting voice recognition subroutine
		VRorder=voicerecognition();
	}
	return 0;
}
```

```
void init(void) {
	//Start Timer
	timer_start();

	// Port A (BUTTONS) initialization
	//Buttons on terminal blocks
	DDRA = 0x00; //Data direction INPUT
	PORTA = 0xFF;	//activate the pull-up resistor for buttons

	// Port B (MOTOR) initialization
	DDRB = 0xFF;	//data direction OUTPUT
	cbi(PORTB,PB4);		//PB4 (OC4) - OC2A (PWM output),
	cbi(MOTOR_PORT,MOTOR_DIRECTION);//PB5 (OC3) - motor direction
	sbi(MOTOR_PORT,MOTOR_ENABLE);	//PB6 (OC2) - enable

	// Port C (LEDS) initialization
	DDRC = 0xFF; //data direction OUTPUT
	PORTC = 0x00;	//LEDS are grounded - 0 means off

	// Port D initialization
	PORTD = 0x00;
	DDRD = 0x00;

	// Port E initialization
	DDRE = 0x00;
	PORTE = 0xFF;	//activate the pull-up resistor for buttons

	//Initialization of the PCB buttons and LED.
	//Set output direction for led, see AT90CAN128 datasheet page 66
	sbi(STATUSLED_DIRECTION,STATUSLED);
	sbi(STATUSLED_PORT,STATUSLED);

	/** Port F initialization*/
	DDRF = 0xFF;
	PORTF = 0x00;

	// Data direction INPUT
	cbi(POTENTIOMETER_DDR,POTENTIOMETER);
	sbi(POTENTIOMETER_PORT,POTENTIOMETER);


	// Port G initialization
	PORTG = 0xFF;
	DDRG = 0xFF;

	TCCR1A = 0;
	TCCR1B = 0;
	TCCR1C = 0;
	TCCR2A = 0;

	//init_pwm(); -- Pulse Width Modulation is not being used in this version

	// Initializing both UARTs
	uartInit();
```

```
	// UART0 used for rprintf function
	rprintfInit(uart0SendByte);


	// Initialization of the potentiometer, interrupts and accelerometer
	init_potentiometer();
	init_interrupt();
	i2cInit();
	lis_init();
	lis_power_up();

	// Initialization of the states in statemachine.c
	next_state = 0;
	goto_init_drive();
}
```

statemachine.h

```
#ifndef STATEMACHINE_H
#define STATEMACHINE_H

// Buttons in A port

//BLACK wire 1
#define BUTTON_CHAIR 0
//GRAY WIRE 1
#define BUTTON_BED 1
#define BUTTON_DOWN 1
//BLUE WIRE 1
#define BUTTON_HOME 2
#define BUTTON_UP 2
//YELLOW WIRE 1
#define BUTTON_RIGHT 3
//RED WIRE 1
#define BUTTON_LEFT 4
//RED WIRE 2
#define BUTTON_AUTO 5
//YELLOW wire 2
#define BUTTON_MANUAL 6
//GREEN WIRE 2
#define BUTTON_TIGHT_LOOSE 7

// Buttons in E port

//GREY WIRE 3
#define AT_HOME 6
//Button on the olimex board
#define ONBOARDBUTTON 5

// Port A buttons
#define BUTTON_PINA PINA
#define BUTTON_PORTA PORTA
#define BUTTON_DDRA DDRA

// Port E buttons
#define BUTTON_PORTE PORTE
#define BUTTON_PINE PINE
#define BUTTON_DDRE DDRE

// Available states in the state machine
#define STATE_NONE 0
#define STATE_LEFT 1
#define STATE_RIGHT 2
#define STATE_MANUAL 3
#define STATE_AUTO 4
#define STATE_TIGHTENED 5
#define STATE_TIGHTENING 6
#define STATE_LOOSENING 7
#define STATE_HOME 8
#define STATE_BED 9
#define STATE_CHAIR 10
```

```
#define STATE_AT_HOME 11
#define STATE_NOT_READY 12
#define STATE_INIT_DRIVE 13
#define STATE_MANUAL_TIGHTENING 14
#define STATE_MANUAL_LOOSENING 15
#define STATE_PUSH_LEFT 16
#define STATE_PUSH_RIGHT 17
#define STATE_MOVING_LEFT 18
#define STATE_MOVING_RIGHT 19

#define NEXT_STATE_NONE 0
#define NEXT_STATE_HOME 1
#define NEXT_STATE_BED 2
#define NEXT_STATE_CHAIR 3
#define NEXT_STATE_MANUAL 4
#define NEXT_STATE_AUTO 5
#define NEXT_STATE_NOT_READY 6

// Definitions to use in combination with is_straight
#define STATIC 0
#define MOVING_LEFT 1
#define MOVING_RIGHT 2
#define PUSHING_LEFT 3
#define PUSHING_RIGHT 4

uint8_t button_changedA(uint8_t button);
uint8_t button_changedE(uint8_t button);
uint8_t button_down_portA(uint8_t button);
uint8_t button_down_portE(uint8_t button);

uint8_t goto_none();
uint8_t goto_auto();
uint8_t goto_manual();
uint8_t goto_left();
uint8_t goto_right();
uint8_t goto_tight();
uint8_t goto_tightened();
uint8_t goto_loose();
uint8_t goto_home();
uint8_t goto_bed();
uint8_t goto_chair();
uint8_t state_machine();
uint8_t at_home();
uint8_t goto_not_ready();
uint8_t goto_init_drive();
uint8_t goto_manual_tight();
uint8_t goto_manual_loose();
uint8_t goto_push_left();
uint8_t goto_push_right();
uint8_t goto_moving_right();
uint8_t goto_moving_left();
#endif
```

statemachine.c

```
/*
 * This file contains all the subroutines managing the large state machine of the
 * system state_machine(int state) function is called by an interrupt (configured
 * in interrupts.c) at a frequency of 1KHz approximately
 *
 * It is the main and most important function in the whole system:
 * checks the state of the system inputs and outputs and manages all the user's
 * instructions and system movement
 *
 */


#include <avr/io.h>
#include "statemachine.h"
#include "leds.h"
#include <util/delay.h>
#include "global.h"
#include "slidepotentiometer.h"
#include "motor.h"
#include "accelerometer.h"
#include "rprintf.h"
#include "interrupts.h"
#include "uart2.h"
#include "voicerecon.h"
#include "canproto.h"

// Voice Recognition order from the main routine
extern unsigned char VRorder;

/**Last state of the buttons. 0 if button is down*/
uint8_t button_stateA;
uint8_t button_stateE;
uint8_t buttons_changedA;
uint8_t buttons_changedE;

static int previous_state=0;
int direction=0;

// Defines the moving way of the shaft motor
uint8_t tightening_time = 0;
uint8_t loosening_time = 0;

// Auto state = 1, Manual state = 2
uint8_t auto_or_manual_state=0;

// Returns the nonzero if specified button is pressed
uint8_t button_down_portA(uint8_t button) {
	return ((~BUTTON_PINA) & (1 << button));
}
uint8_t button_down_portE(uint8_t button) {
	return ((~BUTTON_PINE) & (1 << button));
}
```

```
// Below are defined all the functions that make transitions between states and control
the
// LEDs in the control pad

/* ----- Go to NONE state ----- */
uint8_t goto_none() {
	led_on(LED_READY);
	led_on(LED_ACTION);
	state = STATE_NONE;
	return 0;
}

/* ----- Go to AUTO state ----- */
uint8_t goto_auto() {
	auto_or_manual_state = 1;
	rprintf("AUTO\r\n");
	led_on(LED_READY);
	led_off(LED_ACTION);
	state = STATE_AUTO;
	return 0;
}

/* ----- Go to MANUAL state ----- */
uint8_t goto_manual() {
	auto_or_manual_state = 2;
	rprintf("MANUAL\r\n");
	led_on(LED_READY);
	led_off(LED_ACTION);
	state = STATE_MANUAL;
	return 0;
}

/* ----- Go to LEFT state ----- */
uint8_t goto_left() {
	rprintf("MOVING LEFT...\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_LEFT;
	return 0;
}

/* ----- Go to RIGHT state ----- */
uint8_t goto_right() {
	rprintf("MOVING RIGHT...\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_RIGHT;
	return 0;
}

/* ----- Go to TIGHTENING state ----- */
uint8_t goto_tight() {
	timer_start();
	rprintf("TIGHTENING\r\n");
	led_on(LED_ACTION);
```

```
	led_off(LED_READY);
	state = STATE_TIGHTENING;
	return 0;
}

/* ----- Go to TIGHTENED state ----- */
uint8_t goto_tightened() {
	rprintf("TIGHTENED\r\n");
	led_off(LED_ACTION);
	led_on(LED_READY);
	state = STATE_TIGHTENED;
	return 0;
}

/* ----- Go to LOOSENING state ----- */
uint8_t goto_loose() {
	timer_start();
	rprintf("LOOSENING\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_LOOSENING;
	return 0;
}

/* ----- Go to HOME state ----- */
uint8_t goto_home() {
	rprintf("GOING HOME\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_HOME;
	return 0;
}

/* ----- Go to BED state ----- */
uint8_t goto_bed() {
	rprintf("GOING TO THE BED\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_BED;
	return 0;
}

/* ----- Go to CHAIR state ----- */
uint8_t goto_chair() {
	rprintf("GOING TO THE CHAIR\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_CHAIR;
	return 0;
}

/* ----- Go to AT HOME state ----- */
uint8_t at_home() {
	rprintf("AT HOME\r\n");
	led_on(LED_ACTION);
```

```
	led_on(LED_READY);
	state = STATE_AT_HOME;
	return 0;
}

/* ----- Go to NOT READY state ----- */
uint8_t goto_not_ready() {
	timer_start();
	led_on(LED_ACTION);
	led_on(LED_READY);
	state = STATE_NOT_READY;
	return 0;
}

/* ----- Go to INIT DRIVE state ----- */
uint8_t goto_init_drive() {

	timer_start();
	rprintf("HELLO!\r\n");
	led_on(LED_ACTION);
	led_on(LED_READY);
	state = STATE_INIT_DRIVE;
	return 0;
}

/* ----- Go to MANUAL TIGHTENING state ----- */
uint8_t goto_manual_tight() {
	rprintf("MANUAL TIGHTENING\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_MANUAL_TIGHTENING;
	return 0;
}

/* ----- Go to MANUAL LOOSENING state ----- */
uint8_t goto_manual_loose() {
	rprintf("MANUAL LOOSENING\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_MANUAL_LOOSENING;
	return 0;
}

/* ----- Go to PUSH LEFT (SERVO) state ----- */
uint8_t goto_push_left(){
	rprintf("PUSHING LEFT\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_PUSH_LEFT;
	return 0;
}

/* ----- Go to PUSH RIGHT (SERVO) state ----- */
uint8_t goto_push_right(){
	rprintf("PUSHING RIGHT\r\n");
```

```
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_PUSH_RIGHT;
	return 0;
}

/* ----- Go to MOVING RIGHT (Voice recognition only) state ----- */
uint8_t goto_moving_right() {
	rprintf("MOVING RIGHT...\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_MOVING_RIGHT;
	return 0;
}

/* ----- Go to MOVING LEFT (Voice recognition only) state ----- */
uint8_t goto_moving_left() {
	rprintf("MOVING LEFT...\r\n");
	led_on(LED_ACTION);
	led_off(LED_READY);
	state = STATE_MOVING_LEFT;
	return 0;
}


// Interrupt-driven state machine ruling the system:
// It is a large switch loop checking the current state of the system first, and
// then checks the I/O
uint8_t state_machine() {

	switch (state) {

	// First state of the state machine. Initializes the drive controller
	case STATE_INIT_DRIVE:

		rprintf("INIT DRIVE\r\n");
		drive_init();
		//wait 2 seconds for initializing drive
		while(time()<32000) {
			timer_run();
		}
		rprintf("INIT DRIVE OK\r\n");
		goto_not_ready();
		break;

	// Loosen the support and going home
	case STATE_NOT_READY:

		loosen();
		//wait 10 seconds
		while(time()<64000) {
			timer_run();
		}

		rprintf("NOT TIGHTENED\r\n");
```

```
        rprintf("GOING HOME...\r\n");
        //defines acc
        drive_write_param32(CODE_CACC, ACC_1);
        //set speed mode 1
        drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
        //set command speed to the LEFT
        drive_write_paramf(CODE_CSPD, 80);
        //start motion
        drive_send_command(CODE_UPD);

        //until at home
        while (!button_down_portE(AT_HOME));

        //stop ceiling motor
        drive_send_command(CODE_STOP);
        //reset faults
        drive_send_command(CODE_FAULTRESET);
        //set actual position = 0
        drive_write_param32(CODE_SAP, 0);

        rprintf("AT HOME\r\n");
        goto_none();
        break;
        break;

        // Checking mode switch and configuring the motor
        case STATE_NONE:

        if (button_down_portA(BUTTON_MANUAL)) {
               //reset faults
               drive_send_command(CODE_FAULTRESET);
               //defines acc
               drive_write_param32(CODE_CACC, ACC_1);
               //update values
               drive_send_command(CODE_UPD);

               goto_manual();
               break;

        } else if (button_down_portA(BUTTON_AUTO)) {
               //reset faults
               drive_send_command(CODE_FAULTRESET);
               //defines acc
               drive_write_param32(CODE_CACC, ACC_01);
               //defines speed
               drive_write_paramf(CODE_CSPD, 130);
               //update values
               drive_send_command(CODE_UPD);

               goto_auto();
               break;
        }
        break;

// Manual state --- Buttons used: Left, Right, Tightening, Manual Tightening
```

```
// and Manual Loosening. It also checks accelerometer values for servo
// control
case STATE_MANUAL:

	previous_state=state;
	// Check if any of the listed buttons are pressed and go to the desired
	// state
	if (button_down_portA(BUTTON_LEFT)) {
		goto_left();
		break;
	} else if (button_down_portA(BUTTON_RIGHT)) {
		goto_right();
		break;
	} else if (button_down_portA(BUTTON_TIGHT_LOOSE)) {
		goto_tight();
		break;
	} else if (button_down_portA(BUTTON_AUTO)) {
		goto_none();
		break;
	} else if (button_down_portA(BUTTON_UP)) {
		goto_manual_loose();
		break;
	} else if (button_down_portA(BUTTON_DOWN)) {
		goto_manual_tight();
		break;
	/* ----- Check if it is being pushed (servo control) ----- */
	} else if (is_straight(PUSHING_LEFT)) {
		goto_push_left();
		break;
	} else if (is_straight(PUSHING_RIGHT)) {
		goto_push_right();
		break;
	}
	break;


// Auto state --- Buttons used: Home, Bed, Chair and Tightening.
// It also checks voice recognition commands (VRorder)
case STATE_AUTO:

	if (button_down_portA(BUTTON_MANUAL)) {
		goto_none();
		break;
	} else if ((button_down_portA(BUTTON_HOME))||(VRorder==VR_GO_HOME)) {
		goto_home();
		break;
	} else if ((button_down_portA(BUTTON_BED))||(VRorder==VR_GO_TO_BED)) {
		goto_bed();
		break;
	} else if ((button_down_portA(BUTTON_CHAIR))||(VRorder==VR_GO_TO_CHAIR)) {
		goto_chair();
		break;
	} else if (VRorder==VR_MOVE_LEFT) {
		goto_moving_left();
		break;
```

```
		} else if (VRorder==VR_MOVE_RIGHT) {
			goto_moving_right();
			break;
		} else if ((button_down_portA(BUTTON_TIGHT_LOOSE))||(VRorder==VR_TIGHTEN)){
			goto_tight();
			break;
		}
		break;


	// Moving left in manual mode while Left button is pressed and the movement
	// limits not reached
	case STATE_LEFT:

		if (is_straight(MOVING_LEFT)) {
			//set speed mode 1
			drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
			//set command speed to the LEFT
			drive_write_paramf(CODE_CSPD, 80);
			//start motion
			drive_send_command(CODE_UPD);
			if (!button_down_portA(BUTTON_LEFT) || button_down_portE(AT_HOME) ||
drive_read_param32(ASK_APOS) > 1000) {
				//button is not pressed anymore, STOP the ceiling motor
				drive_send_command(CODE_STOP);
				goto_manual();
				break;
			}
		} else {
			drive_send_command(CODE_STOP);
			goto_manual();
			break;
		}

		break;

	// Moving right in manual mode while Right button is pressed and the movement
	// limits not reached
	case STATE_RIGHT:

		if (is_straight(MOVING_RIGHT)) {
			//set speed mode 1
			drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
			//set command speed to the RIGHT
			drive_write_paramf(CODE_CSPD, -60);
			//start motion
			drive_send_command(CODE_UPD);

			if (!button_down_portA(BUTTON_RIGHT) || drive_read_param32(ASK_APOS) <=
-6250000 ) {
				//button is not pressed anymore, STOP the ceiling motor
				drive_send_command(CODE_STOP);
				goto_manual();
				break;
			}
```

```
		}
		else
		{
			drive_send_command(CODE_STOP);
			goto_manual();
			break;
		}
		break;

	/* VR instruction: Move left. The support will only stop if MANUAL mode is
	selected, it is given a different VR or AUTO instruction or it reaches the
	boundaries */
	case STATE_MOVING_LEFT:
	{
		if (is_straight(MOVING_LEFT)) {
			//set speed mode 1
			drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
			//set command speed to the LEFT
			drive_write_paramf(CODE_CSPD, 80);
			//start motion
			drive_send_command(CODE_UPD);

			//check if home, bed or chair is pressed (or manual mode)
			if (button_down_portA(BUTTON_MANUAL)) {
				drive_send_command(CODE_STOP);
				goto_none();
				break;
			}
			else if ((button_down_portA(BUTTON_HOME))||(VRorder==VR_GO_HOME)) {
				//start to move the ceiling motor to position HOME
				drive_send_command(CODE_STOP);
				goto_home();
				break;
			} else if ((button_down_portA(BUTTON_BED))||(VRorder==VR_GO_TO_BED)) {
				//start to move the ceiling motor to position BED
				drive_send_command(CODE_STOP);
				goto_bed();
				break;
			} else if
((button_down_portA(BUTTON_CHAIR))||(VRorder==VR_GO_TO_CHAIR)) {
				//start to move the ceiling motor to position CHAIR
				drive_send_command(CODE_STOP);
				goto_chair();
				break;
			} else if (VRorder==VR_MOVE_RIGHT){
				drive_send_command(CODE_STOP);
				goto_moving_right();
				break;
				/* If VR or manually order to start tightening, stop the
				ceiling motor and start tightening */
			} else if
((VRorder==VR_TIGHTEN)||(button_down_portA(BUTTON_TIGHT_LOOSE))){
				drive_send_command(CODE_STOP);
				goto_tight();
				break;
```

```
                        /* If VR ordered to stop or it reaches the limits, stop the
                     ceiling motor */
                  } else if ((drive_read_param32(ASK_APOS) > 1000)||(VRorder==VR_STOP)) {
                        drive_send_command(CODE_STOP);
                        goto_auto();
                        break;
                  }
            } else {
                  // The support is leaned, don't move
                  drive_send_command(CODE_STOP);
                  goto_auto();
                  break;
            }
            break;
      }
      case STATE_MOVING_RIGHT:
      {
            /* VR instruction: Move right. The support will only stop if MANUAL
mode is selected, it is given a different VR or AUTO instruction or             it reaches
the boundaries */
            if (is_straight(MOVING_RIGHT))
            {
                  // Set speed mode 1
                  drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
                  // Set command speed to the RIGHT
                  drive_write_paramf(CODE_CSPD, -60);
                  // Start motion
                  drive_send_command(CODE_UPD);

                  // Check if home, bed or chair is pressed (or manual mode)
                  if (button_down_portA(BUTTON_MANUAL)) {
                        drive_send_command(CODE_STOP);
                        goto_none();
                        break;
                  }
                  else if ((button_down_portA(BUTTON_HOME))||(VRorder==VR_GO_HOME)) {
                        // Start to move the ceiling motor to position HOME
                        drive_send_command(CODE_STOP);
                        goto_home();
                        break;
                  } else if ((button_down_portA(BUTTON_BED))||(VRorder==VR_GO_TO_BED)) {
                        // Start to move the ceiling motor to position BED
                        drive_send_command(CODE_STOP);
                        goto_bed();
                        break;
                  } else if
((button_down_portA(BUTTON_CHAIR))||(VRorder==VR_GO_TO_CHAIR)) {
                        // Start to move the ceiling motor to position CHAIR
                        drive_send_command(CODE_STOP);
                        goto_chair();
                        break;
                  } else if (VRorder==VR_MOVE_LEFT){
                        drive_send_command(CODE_STOP);
                        goto_moving_left();
                        break;
```

```
                        /* If VR or manually order to start tightening, stop the
                     ceiling motor and start tightening */
                  } else if
((VRorder==VR_TIGHTEN)||(button_down_portA(BUTTON_TIGHT_LOOSE))){
                        drive_send_command(CODE_STOP);
                        goto_tight();
                        break;
                        /* If VR ordered to stop or it reaches the limits, stop the
                     ceiling motor */
                  } else if ((drive_read_param32(ASK_APOS) <= -
6250000)||(VRorder==VR_STOP)) {
                        drive_send_command(CODE_STOP);
                        goto_auto();
                        break;
                  }
            } else {
                  // The support is leaned, don't move
                  drive_send_command(CODE_STOP);
                  goto_auto();
                  break;
            }
            break;
      }

      /* Auto instruction: Go home. The support will move to the defined home
       * position, it will stop if another valid instruction (buttons or VR
       * commands) is given */
      case STATE_HOME:
            previous_state=state;
            if (is_straight(MOVING_LEFT)) {

                  // Position absolute
                  drive_write_param32(CODE_MODE_TYPE, CODE_CPA);
                  // Sets command position = 0
                  drive_write_param32(CODE_CPOS, 1000);
                  // Sets position mode 3
                  drive_write_param32(CODE_MODE_TYPE, CODE_PP3);
                  // Keep position and speed reference
                  drive_write_param32(CODE_MODE_TYPE, CODE_TUM1);
                  // Start motion
                  drive_send_command(CODE_UPD);

                  if (button_down_portA(BUTTON_TIGHT_LOOSE)) {
                        drive_send_command(CODE_STOP);
                        goto_tight();
                        break;
                  } else if (button_down_portA(BUTTON_CHAIR)){
                        drive_send_command(CODE_STOP);
                        goto_chair();
                  } else if (button_down_portA(BUTTON_BED)){
                        drive_send_command(CODE_STOP);
                        goto_bed();
                  }
                  if (button_down_portA(BUTTON_MANUAL)) {
                        drive_send_command(CODE_STOP);
```

```
                goto_manual();
                break;
        } else if (button_down_portE(AT_HOME) ) {
                drive_send_command(CODE_STOP);
                at_home();
                break;
        }
        // Check VR commands
        switch (VRorder)
        {
                case VR_MOVE_LEFT:
                {
                        drive_send_command(CODE_STOP);
                        goto_moving_left();
                        break;
                }
                case VR_MOVE_RIGHT:
                {
                        drive_send_command(CODE_STOP);
                        goto_moving_right();
                        break;
                }
                case VR_STOP:
                {
                        drive_send_command(CODE_STOP);
                        goto_auto();
                        break;
                }
                case VR_TIGHTEN:
                {
                        drive_send_command(CODE_STOP);
                        goto_tight();
                        break;
                }
                case VR_GO_TO_CHAIR:
                {
                        drive_send_command(CODE_STOP);
                        goto_chair();
                        break;
                }
                case VR_GO_TO_BED:
                {
                        drive_send_command(CODE_STOP);
                        goto_bed();
                        break;
                }
        }
} else {
        drive_send_command(CODE_STOP);
        goto_auto();
        break;
}
break;

/* Auto instruction: Go to bed. The support will move to the defined bed
```

```
 * position, it will stop if another valid instruction (buttons or VR
 * commands) is given */
case STATE_BED:

      if (previous_state==STATE_HOME){
            direction=MOVING_RIGHT;
      } else if (previous_state==STATE_CHAIR){
            direction=MOVING_LEFT;
      } else {
            direction=STATIC;
      }

      if (is_straight(direction)) {
            // Position absolute
            drive_write_param32(CODE_MODE_TYPE, CODE_CPA);
            // Sets command position = -3520548
            drive_write_param32(CODE_CPOS, -3520548);
            // Sets position mode 3
            drive_write_param32(CODE_MODE_TYPE, CODE_PP3);
            // Keep position and speed reference
            drive_write_param32(CODE_MODE_TYPE, CODE_TUM1);
            // Start motion
            drive_send_command(CODE_UPD);

            if (button_down_portA(BUTTON_TIGHT_LOOSE)) {
                  drive_send_command(CODE_STOP);
                  goto_tight();
                  break;
            } else if (button_down_portA(BUTTON_HOME)){
                  drive_send_command(CODE_STOP);
                  goto_home();
            } else if (button_down_portA(BUTTON_CHAIR)){
                  drive_send_command(CODE_STOP);
                  goto_chair();
            }
            if (button_down_portA(BUTTON_MANUAL)) {
                  drive_send_command(CODE_STOP);
                  goto_manual();
                  break;
            }
            if (drive_read_param32(ASK_APOS) == drive_read_param32(ASK_CPOS)) {
                  goto_tight();
                  break;
            }

            // Check VR commands
            switch (VRorder)
            {
                  case VR_MOVE_LEFT:
                  {
                        drive_send_command(CODE_STOP);
                        goto_moving_left();
                        break;
                  }
                  case VR_MOVE_RIGHT:
```

```
                        {
                                drive_send_command(CODE_STOP);
                                goto_moving_right();
                                break;
                        }
                        case VR_STOP:
                        {
                                drive_send_command(CODE_STOP);
                                goto_auto();
                                break;
                        }
                        case VR_TIGHTEN:
                        {
                                drive_send_command(CODE_STOP);
                                goto_tight();
                                break;
                        }
                        case VR_GO_HOME:
                        {
                                drive_send_command(CODE_STOP);
                                goto_home();
                                break;
                        }
                        case VR_GO_TO_CHAIR:
                        {
                                drive_send_command(CODE_STOP);
                                goto_chair();
                                break;
                        }
                }
        } else {
                drive_send_command(CODE_STOP);
                goto_auto();
                break;
        }
        break;

    /* Auto instructions: Go to chair. The support will move to the defined chair
     * position, it will stop if another valid instruction (buttons or VR
* commands) is given */
    case STATE_CHAIR:
        previous_state=state;
        if (is_straight(MOVING_RIGHT)) {
                // Position absolute
                drive_write_param32(CODE_MODE_TYPE, CODE_CPA);
                // Sets command position = -5867580
                drive_write_param32(CODE_CPOS, -5867580);
                // Sets position mode 3
                drive_write_param32(CODE_MODE_TYPE, CODE_PP3);
                // Keep position and speed reference
                drive_write_param32(CODE_MODE_TYPE, CODE_TUM1);
                // Start motion
                drive_send_command(CODE_UPD);

                if (button_down_portA(BUTTON_TIGHT_LOOSE)) {
```

```
		drive_send_command(CODE_STOP);
		goto_tight();
		break;
	} else if (button_down_portA(BUTTON_HOME)){
		drive_send_command(CODE_STOP);
		goto_home();
	} else if (button_down_portA(BUTTON_BED)){
		drive_send_command(CODE_STOP);
		goto_bed();
	}
	if (button_down_portA(BUTTON_MANUAL)) {
		drive_send_command(CODE_STOP);
		goto_manual();
		break;
	}
	if (drive_read_param32(ASK_APOS) == drive_read_param32(ASK_CPOS)) {
		goto_tight();
	}
	// Check VR commands
	switch (VRorder)
	{
		case VR_MOVE_LEFT:
		{
			drive_send_command(CODE_STOP);
			goto_moving_left();
			break;
		}
		case VR_MOVE_RIGHT:
		{
			drive_send_command(CODE_STOP);
			goto_moving_right();
			break;
		}
		case VR_STOP:
		{
			drive_send_command(CODE_STOP);
			goto_auto();
			break;
		}
		case VR_TIGHTEN:
		{
			drive_send_command(CODE_STOP);
			goto_tight();
			break;
		}
		case VR_GO_HOME:
		{
			drive_send_command(CODE_STOP);
			goto_home();
			break;
		}
		case VR_GO_TO_BED:
		{
			drive_send_command(CODE_STOP);
			goto_bed();
```

```
                        break;
                    }
                }
            } else {
                drive_send_command(CODE_STOP);
                goto_auto();
                break;
            }
            break;

    case STATE_AT_HOME:

            if (drive_read_param32(ASK_APOS) != drive_read_param32(ASK_CPOS)) {
                // Wait until actual speed = 0
                while (drive_read_param32(ASK_ASPD) != 0)
                    ;

                // Reset faults
                drive_send_command(CODE_FAULTRESET);
                // Set actual position = 0
                drive_write_param32(CODE_SAP, 0);
                // Update
                drive_send_command(CODE_UPD);

                goto_auto();
                break;
            } else {
                drive_send_command(CODE_FAULTRESET);
                goto_auto();
                break;
            }
            break;

    // Tighten the support pole until it is steady and go to state TIGHTENED.
    // The operation cannot be stopped
    case STATE_TIGHTENING:
            if (is_straight(STATIC)) {
                tighten();
                tightening_time = time();
                loosening_time = tightening_time + 5;

                if (button_down_portA(BUTTON_TIGHT_LOOSE) || !is_straight(STATIC)) {

                    if (auto_or_manual_state == 1) {
                        next_state = NEXT_STATE_AUTO;
                        stop_tightening_motor();
                        goto_loose();
                        break;
                    } else if (auto_or_manual_state == 2) {
                        next_state = NEXT_STATE_MANUAL;
                        stop_tightening_motor();
                        goto_loose();
                        break;
                    }
                }
```

```
                        if (is_steady()) {
                                goto_tightened();
                                stop_tightening_motor();
                                break;
                        }
                } else {
                        drive_send_command(CODE_STOP);
                        goto_auto();
                        break;
                }
                break;

        // Loosen the support pole a fixed time (5 seconds more than the tightening  //
time)
        case STATE_LOOSENING:
                loosen();
                loosening_time--;
                rprintf("TIME: %d\r\n",loosening_time);

                if (loosening_time == 0) {

                        stop_tightening_motor();

                        if (next_state == NEXT_STATE_HOME) {
                                goto_home();
                                break;
                        }
                        if (next_state == NEXT_STATE_BED) {
                                goto_bed();
                                break;
                        }
                        if (next_state == NEXT_STATE_CHAIR) {
                                goto_chair();
                                break;
                        }
                        if (next_state == NEXT_STATE_MANUAL) {
                                goto_manual();
                                break;
                        }
                        if (next_state == NEXT_STATE_AUTO) {
                                goto_auto();
                                break;
                        }
                        if (next_state == NEXT_STATE_NOT_READY) {
                                goto_not_ready();
                                break;
                        }
                        break;
                }
                break;

        case STATE_TIGHTENED:
                if (((VRorder==VR_GO_HOME)||(button_down_portA(BUTTON_HOME))) &&
auto_or_manual_state == 1) {
```

```
                        next_state = NEXT_STATE_HOME;
                        goto_loose();
                        break;
                }
                if (((VRorder==VR_GO_TO_BED)||(button_down_portA(BUTTON_BED))) &&
auto_or_manual_state == 1) {
                        next_state = NEXT_STATE_BED;
                        goto_loose();
                        break;
                }
                if (((VRorder==VR_GO_TO_CHAIR)||(button_down_portA(BUTTON_CHAIR))) &&
auto_or_manual_state == 1) {
                        next_state = NEXT_STATE_CHAIR;
                        goto_loose();
                        break;
                }
                if ((button_down_portA(BUTTON_UP))&&(auto_or_manual_state == 2)) {
                        goto_manual_tight();
                }
                if ((VRorder==VR_TIGHTEN)&&(auto_or_manual_state == 1))
                {
                        next_state = NEXT_STATE_AUTO;
                        goto_loose();
                        break;
                }
                if (button_down_portA(BUTTON_TIGHT_LOOSE)) {

                        if (auto_or_manual_state == 1) {
                                next_state = NEXT_STATE_AUTO;
                                goto_loose();
                                break;
                        } else if (auto_or_manual_state == 2) {
                                next_state = NEXT_STATE_MANUAL;
                                goto_loose();
                                break;
                        }
                }
                break;

        // Loose the support pole while the Manual Loosening button is pressed
        case STATE_MANUAL_LOOSENING:
                if (is_straight(STATIC)&&(button_down_portA(BUTTON_UP))){
                        loosen();
                } else {
                        goto_manual();
                        stop_tightening_motor();
                }
                break;

        // Tighten the support pole while the Manual Tightening button is pressed and
        // the pole is not steady
        case STATE_MANUAL_TIGHTENING:
                if (is_straight(STATIC)&&(button_down_portA(BUTTON_DOWN))){
                        tighten();
                        if (is_steady()){
```

```
                        goto_tightened();
                        stop_tightening_motor();
                        break;
                }
        } else {
                goto_manual();
                stop_tightening_motor();
        }
        break;

    // If the pole is leaned to the left, move it
    case STATE_PUSH_LEFT:
        if ((is_straight(PUSHING_LEFT))&&(drive_read_param32(ASK_APOS <= 1000))){
        // Set speed mode 1
        drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
        // Set command speed to the LEFT
        drive_write_paramf(CODE_CSPD, 60);
        // Start motion
        drive_send_command(CODE_UPD);
        } else{
                /* Button is not pressed anymore, the support is no straight or
         * it has reached the end, STOP the ceiling motor*/
                drive_send_command(CODE_STOP);
                goto_manual();
                break;
        }
        break;

    // If the support is leaned to the right, move it
    case STATE_PUSH_RIGHT:
        if ((is_straight(PUSHING_RIGHT))&&(drive_read_param32(ASK_APOS) > -6250000
))){
                // Set speed mode 1
                drive_write_param32(CODE_MODE_TYPE, CODE_SP1);
                // Set command speed to the RIGHT
                drive_write_paramf(CODE_CSPD, -60);
                // Start motion
                drive_send_command(CODE_UPD);
        } else {
                        /* Button is not pressed anymore, the support is no
                         * straight or it has reached the end, STOP the ceiling
                 * motor*/
                        drive_send_command(CODE_STOP);
                        goto_manual();
                        break;
        }
        break;
    }
    return 0;
}
```

accelerometer.h

```
#ifndef ACCELEROMETER_H
#define ACCELEROMETER_H

void readAcceleration(void);
int is_straight(int state);
void tellAcceleration(uint16_t *x, uint16_t *y, uint16_t *z);

typedef struct {
	uint16_t x;
	uint16_t y;
	uint16_t z;
} Tacceleration;

void initAcc();
void accTOverflow();

uint16_t getX();
uint16_t getY();
uint16_t getZ();


#endif //ACCELEROMETER_H
```

accelerometer.c

```
/* In this file all the accelerometer routines are defined:
 * Leaning control, collision detection and servo control.
 */

#include "global.h"
#include <avr/interrupt.h>
#include <avr/pgmspace.h>
#include "accelerometer.h"
#include "lis3lv02_driver.h"
#include "i2c.h"
#include "leds.h"
#include "timer.h"
#include "rprintf.h"
#include "statemachine.h"

/* Definition of the trigger values of the accelerometer and used to determine if
 * the pole is straight enough for continuing its move, or if it is being pushed.
 *
 * Accelerometer values are 12-bit integers (0 - 4095)
 */

// Static values
#define MIN_X 10
#define MAX_X 100
#define MIN_Y 40
#define MAX_Y 100
#define MIN_Z 4075
#define MAX_Z 20

/* X axis values when the support pole is moving leftwards, rightwards or being
pushed X axis is parallel to the support rail */
#define MIN_X_LEFT 10
#define MAX_X_LEFT 100
#define MIN_X_RIGHT 1
#define MAX_X_RIGHT 100
#define MIN_X_PUSH_LEFT 4065
#define MAX_X_PUSH_LEFT 20
#define MIN_X_PUSH_RIGHT 80
#define MAX_X_PUSH_RIGHT 220

int min_x;
int max_x;
Tacceleration acceleration;
// int i=0; Auxiliar variable used when monitoring accelerometer values using
// readAcceleration() function

// Accelerometer initialization subroutine
void initAcc() {
      i2cInit();
      lis_init();
      lis_power_up();
}
```

```
// Read each axis values using I2C protocol
uint16_t getX() {
	return lis_read_x();
}
uint16_t getY() {
	return lis_read_y();
}
uint16_t getZ() {
	return lis_read_z();
}


// Read and save accelerometer values using I2C bus
void readAcceleration(void) {

	unsigned char buffer[XYZ_BUF_SIZE];

	lis_read_xyz_b(buffer);
	acceleration.x = buffer[0] | buffer[1] << 4;
	acceleration.y = buffer[2] | buffer[3] << 4;
	acceleration.z = buffer[4] | buffer[5] << 4;
/* Printing for testing purposes only
 *	if (i==10){
 *		rprintf("x: %d\t\t y: %d\t\t z: %d\r\n", acceleration.x,
 *		acceleration.y, acceleration.z);
 *		i=0;
 *	}
 *	i++;
 */
}

/* is_straight(int state)
 * state: Argument given by state_machine() routine, indicating the movement of the
 * support pole
 * 0:Static -- 1: Moving left -- 2: Moving right -- 3:Pushed left -- 4:Pushed right
 *
 * This function reads the accelerometer values and compares them with the defined
 * limits for the support in its current state of movement. As the movement is only
 * in one direction (left or right) only the value of X axis (parallel to the rail)
 * has to be modified.
 *
 * Returns 1 if the pole is straight enough (Movement commands) or if in manual
 * mode is pushed (Servo control).
 * Returns 0 otherwise
 */
int is_straight(int state){

	readAcceleration();

	// Defining the trigger values for the accelerometer
	switch (state)	{
		case 0:
			min_x = MIN_X;
			max_x = MAX_X;
```

```
                break;

        case 1:
                min_x = MIN_X_LEFT;
                max_x = MAX_X_LEFT;
                break;

        case 2:
                min_x = MIN_X_RIGHT;
                max_x = MAX_X_RIGHT;
                break;

        case 3:
                min_x = MIN_X_PUSH_LEFT;
                max_x = MAX_X_PUSH_LEFT;
                break;

        case 4:
                min_x = MIN_X_PUSH_RIGHT;
                max_x = MAX_X_PUSH_RIGHT;
                break;
    }

    // Determining the state of the support. "Pushed left" has to be separated   //
because of the nature of its X axis values.
    if (state == 3){
        if (((acceleration.x >= min_x) || (acceleration.x <= max_x)) &&
((acceleration.y >= MIN_Y) || (acceleration.y <= MAX_Y)) && ((acceleration.z >= MIN_Z) ||
(acceleration.z <= MAX_Z))){
                return 1;
        } else {
                return 0;
        }
    } else {
        if (((acceleration.x >= min_x) && (acceleration.x <= max_x)) &&
((acceleration.y >= MIN_Y) || (acceleration.y <= MAX_Y)) && ((acceleration.z >= MIN_Z) ||
(acceleration.z <= MAX_Z))){
                return 1;
        } else {
                return 0;
        }
    }
}

// Save accelerometer values in memory
void tellAcceleration(uint16_t *x, uint16_t *y, uint16_t *z) {

    *x = acceleration.x;
    *y = acceleration.y;
    *z = acceleration.z;
}
```

voicerecon.h

```
#ifndef VOICERECON_H_
#define VOICERECON_H_

/* The following definitions help to work with the VRBot module environment
 * Further details of the VRbot commands and responses can be found in
 * the Appendix C of this project */

#define      VR_START_RECOGNITION 'd'
#define VR_MOVE_LEFT 'A'
#define VR_MOVE_RIGHT  'B'
#define VR_STOP 'C'
#define VR_TIGHTEN 'D'
#define VR_GO_HOME 'E'
#define VR_GO_TO_BED 'F'
#define VR_GO_TO_CHAIR 'G'
#define VR_ACK ' '
#define VR_START_RECON 'd'
#define VR_TRIGGER_RECON 'E'
#define VR_ORDER_RECON 'B'
#define      VR_TIMEOUT 't'
#define VR_RECON_OK 'r'
#define VR_ASK_ID 'x'
#define VR_ID0 'A'
#define VR_WAKE 'b'
#define VR_LANGUAGE_SET 'l'
#define VR_ENGLISH 'A'
#define VR_OK 'o'
#define VR_THREE_SECONDS 'F'
#define VR_SET_TIMEOUT 'o'
#define VR_SET_RECOG_LEVEL 'v'
#define VR_SD_EASY 'B'

unsigned char voicerecognition (void);
int VRinitialization(void);
unsigned char SendInstruction(unsigned char instruction, unsigned char response, int
data);
void GetInstruction(unsigned char response);


#endif /* VOICERECON_H_ */
```

voicerecon.c

```
/*
 * All the subroutines managing VRbot module are here
 */
#include "voicerecon.h"
#include "uart2.h"
#include "rprintf.h"

unsigned char response;

unsigned char voicerecognition (void)
{
	int data=0;
	response=0;
	static int initOK;

	// Run initialization subroutine until the device is initialized properly
	while (initOK!=2)
	{
		initOK=VRinitialization();
	}

	/* ----- Asking for the trigger word (Wordset 4 in VRbot) ----- */
	rprintf("Trigger Word: ");
	uart1SendByte(VR_START_RECON);
	response=SendInstruction(VR_TRIGGER_RECON,response,data);
	rprintf("Response: %c ", response);  // Testing purposes

	/* If the trigger word is recognized, start command recognition
	 * if not, print a message (error or timeout)
	 */
	if (response==VR_RECON_OK)
	{
		/* ----- Asking for commands (Wordset 1 in VRbot) ----- */
		rprintf("ACCEPTED -- Instruction: ");
		uart1SendByte(VR_START_RECON);
		response=SendInstruction(VR_ORDER_RECON,response,data);
		rprintf("Response: %c ", response);  // Testing purposes
		if (response==VR_RECON_OK) {
			response=SendInstruction (VR_ACK,response,data);
			rprintf("Response: %c\n", response);  // Testing purposes
		}
		else if (response==VR_TIMEOUT) {
			rprintf("Timeout \n");
		} else {
			rprintf("Recognition failed\n");
		}
	}
	else if (response==VR_TIMEOUT) {
		rprintf("Timeout \n");
	} else {
		rprintf("Recognition failed\n");
	}
	return response;
```

```
}

// Initialization of VRbot module
int VRinitialization(void)
{
	unsigned char response=0;
	int i=0;
	int initOK=1;
	int data=0;
	rprintf("Initializing device\n");

	/* ----- Waking up device from low power/saving mode ----- */
	while ((response != VR_OK)&&(i!=101)) {
		response = SendInstruction (VR_WAKE,response,data);
		if (i==100) {
			rprintf("ERROR: Failed to initialize device\n");
			initOK=0;
		}
		i++;
	}

	/* ----- Ask firmware ID ----- */
	rprintf("Response: %c\n",response);
	response = SendInstruction(VR_ASK_ID,response,data);
	rprintf("Response: %c\n",response);
	if (response != VR_ASK_ID) {
		rprintf("ERROR: Cannot read device ID\n");
		initOK=0;
	}

	/* ----- Send ack and read status (expecting ID=0) ----- */
	response = SendInstruction (VR_ACK,response,data);
	rprintf("Response: %c\n",response);
	if (response != VR_ID0) {
		rprintf("Device ID: %c\n", response);
		rprintf("ERROR: Device ID doesn't match\n");
		initOK=0;
	} else {
		rprintf("Device identified\n");
	}

	/* ----- Set language for built-in system instructions (English) ----- */
	uart1SendByte(VR_LANGUAGE_SET);
	response=SendInstruction (VR_ENGLISH,response,data);
	rprintf("Response: %c\n",response);
	if (response != VR_OK) {
		rprintf("ERROR: Could not set default language for built-in instructions
(English)\n");
		initOK=0;
	} else {
		rprintf("Language set: English\n");
	}

	/* ----- Set timeout (3 seconds) ----- */
	uart1SendByte(VR_SET_TIMEOUT);
```

```
	response=SendInstruction (VR_THREE_SECONDS,response,data);
	if (response != VR_OK) {
		rprintf("ERROR: Timeout not set\n");
		initOK=0;
	} else {
		rprintf("Timeout set: 3 seconds\n");
	}

	/* ----- Set SD recognition level to easy ----- */
	uart1SendByte(VR_SET_RECOG_LEVEL);
	response=SendInstruction(VR_SD_EASY,response,data);
	rprintf("Response: %c\n",response);
	if (response != VR_OK) {
			rprintf("ERROR: Could not set SD recognition level to easy\n");
			initOK=0;
	} else {
		rprintf("Set SD recognition level to easy\n");
	}

	/* ----- Checking if the device has been initialized properly ----- */
	if (initOK==1) {
		initOK=2;
		rprintf("Device initialized\n");
	} else {
		rprintf("DEVICE NOT INITIALIZED\n");
	}
	rprintf("Init ok: %d\n",initOK);
	return initOK;
}

/* SendInstruction (unsigned char instruction, unsigned char response, int data)
 *     instruction: Command (byte) sent to the device through the serial port
 *     data: Byte read from the serial port buffer, -1 if nothing is read from the
 *     serial port response: Valid response from the device
 *
 *     This function sends an instruction to the device and then waits for
 *     a response from it
 */
unsigned char SendInstruction (unsigned char instruction, unsigned char response, int
data)
{
	uart1SendByte(instruction);
	while ((data==-1)||(data==0))
	{
		data=uart1GetByte();
	}
	response=data;
	return response;
}
```

motor.h

```
#ifndef MOTOR_H
#define MOTOR_H

void tighten();
void loosen();
void stop_tightening_motor();
void init_pwm();

#endif
```

motor.c

```
/*
 * In this file the control routines for using the tightening motor are defined
 *
 * The motor is connected to pins 4,5 and 6 of B port of the microcontroller
 * Pin 4: Motor power, PWM could be used here
 * Pin 5: Motor direction (1: Tighten, 0: Loosen)
 * Pin 6: Motor enable
 */


#include <avr/io.h>
#include "util.h"
#include "motor.h"
#include "global.h"
#include "leds.h"

/*
 * Timer2
 *
 * Register: TCCR2A
 * Force output: FOC2A 1
 * Fast PWM mode: WGM21:0 = 3)
 * Setting the COM2A1:0 bits to two will produce a non-inverted PWM
 * 8khz: CS22 CS21 CS20: 0 1 0 clkT2S/8 (From prescaler)
 * COM2A1:0
 */

//Pulse Width Modulation is not in use in this version
void init_pwm() {
	TCCR2A = 0;
	sbi(TCCR2A,FOC2A);  //Force output compare match

	// Fast-pwm mode
	sbi(TCCR2A,WGM21);
	sbi(TCCR2A,WGM20);

	//8-prescaler
	sbi(TCCR2A,CS21);
```

```
	// Non-inverted mode, only when the motor is running
	// sbi(TCCR2A,COM2A1);

	// Set output power; max = 0xFF,
	OCR2A = 0xFF;
}

void tighten() {
	sbi(MOTOR_PORT, MOTOR_DIRECTION);	// Set direction - Tightening
	sbi(PORTB,PB4);				// Instead of PWM, full power
}

void loosen() {
	cbi(MOTOR_PORT, MOTOR_DIRECTION);	// Set direction - Loosening
	sbi(PORTB,PB4);				// Instead of PWM, full power
}

void stop_tightening_motor() {
	cbi(PORTB,PB4);		// Turn off power
}
```

leds.h

```
#ifndef LEDS_H
#define LEDS_H

// Set specified led on
void led_on(uint8_t led);
// Set specified led off
void led_off(uint8_t led);
// Return non-zero if specified led is on
uint8_t led_is_on(uint8_t led);

#define LED_PORT PORTC
#define LED_ACTION 1
#define LED_READY 0

// Status led on the Olimex board
#define STATUSLED 4
#define STATUSLED_PORT PORTE
#define STATUSLED_DIRECTION DDRE

#endif
```

leds.c

```
/*
 * Functions used to manage the LEDs in the control pad
 */
#include <avr/io.h>
#include "util.h"
#include "a2d.h"
#include "i2cconf.h"
#include "i2c.h"
#include "lis3lv02_driver.h"
#include "slidepotentiometer.h"
#include "accelerometer.h"
#include "interrupts.h"
#include "leds.h"
#include "global.h"

void led_on(uint8_t led) {
	sbi(LED_PORT,led);
}
void led_off(uint8_t led) {
	cbi(LED_PORT,led);
}
uint8_t led_is_on(uint8_t led) {
	return ((~LED_PORT) & (1 << led));
}
```

interrupts.h

```
#ifndef SUSPENSION_H
#define SUSPENSION_H

/* Initialize interrupt */
void init_interrupt();
/* Start timer */
void timer_run();
/* Reset timer */
void timer_start();
/* Return value of timer */
uint16_t time();
#endif
```

interrupts.c

```
/* In this file a system interruption is configured to call state_machine()
 * function at a 1KHz frequency
 */
#include <avr/io.h>
#include <avr/interrupt.h>
#include "motor.h"
#include "statemachine.h"
#include "leds.h"
#include "slidepotentiometer.h"
#include "accelerometer.h"
#include "util.h"
#include "interrupts.h"
#include "rprintf.h"
#include "global.h"

uint8_t scaler;
uint16_t interrupt_counter;
volatile uint8_t sec_counter;
volatile uint8_t sec_max;
volatile uint8_t timer_ms;
volatile uint32_t timer;

void timer_run() {
	timer_ms++;
	if (timer_ms > 10) {
		timer_ms = 0;
		timer++;
	}
}

void timer_start() {
	timer_ms = 0;
	timer = 0;
}

uint16_t time() {
```

```
	return timer;
}
/*
 Control Register: TCCR0A
 bits:
 1024 prescaler (~16khz): CS02 CS01 CS00 = 1 0 1
 Normal mode (WGM01:0 = 0).

 Register: TIMSK0
 Enable Interrupt bit: TOIE0
*/
void init_interrupt() {

	// Timer0 is used as an interrupt source at ~16kHz frequency
	TIMSK0 = 0;
	TCCR0A = 0;

	sbi(TIMSK0,TOIE0);
	sbi(TCCR0A,CS02);
	sbi(TCCR0A,CS00);

	sec_counter = 0;
	interrupt_counter = 0;
	scaler = 0;

	sei();
}

// Interrupt frequency 16KHz
ISR(TIMER0_OVF_vect)
{
	timer_run();
	if (scaler > 16)
	{
		// Calling state_machine() function at ~1ms intervals
		state_machine();
		scaler = 0;
	}
	scaler++;
}
```

slidepotentiometer.h

```
#ifndef SLIDEPOTENTIOMETER_H
#define SLIDEPOTENTIOMETER_H

void init_potentiometer(void);
int is_steady(void);
void readPotentiometerValue(void);

#endif
```

slidepotentiometer.c

```
/*
 * In this file all the sliding potentiometer routines are defined
 *
 * The slide potentiometer is attached to the string in the tightening motor
 * and measures if it is steady enough when tightening
 *
 */

#include <avr/io.h>
#include "a2d.h"
#include "inttypes.h"
#include "motor.h"
#include "uart2.h"
#include "rprintf.h"

#define LIMIT_VALUE 980

uint16_t result;
u08 potentio_tick = 0;

void init_potentiometer(void) {
	a2dInit();   //Initialize potentiometer - a2d.c
}

int is_steady(void)
{
	// Read analog value from A/D channel 7
	// Result is 10-bit integer - max value is 1024
	// Analog to Digital conversion is Slow and this call blocks until
	// the value is ready. This shouldn't be run inside an interrupt*/
	result = a2dConvert10bit(0);

	if (result < LIMIT_VALUE) {
		return 1;

	} else
		return 0;
}

void readPotentiometerValue(void)
```

```
{
	result = a2dConvert10bit(0);
	// Printing for testing
	if ((potentio_tick % 160000) == 0) {
		rprintf("POTENTIOMETER: %d\r\n",result);

	}
	potentio_tick++;

}
```

lis3l02_driver.h

```
#ifndef LIS3LV02DRIVER_H_
#define LIS3LV02DRIVER_H_

#include <inttypes.h>

/*
 *     Converted to use i2c bus, not tested yet, probably not yet working
 */

/* The id for the micro bus.
 * Note that in the bus_select function the CS lines will be set to this number,
 * and in the bus_unselect they will all be put to 0. As we don't use a decoder
 * yet, we connect the CS3 bus line to the Chip Select input of the accelerometer.
 * As this input is active low, the number 7 selects (puts to 0) the CS3 signal
 * only, and therefore enables the accelerometer. This is a temporary trick until
 * we put a decoder for the CS bus lines!!!
 */
//#define LIS_BUS_ID 7
//#define LIS_BUS_ID 8

#define LIS_ADDR 0x3a

#define portCHAR unsigned char

//The following defines are byte bit masks to select read/write operations with/without
address autoincrement
//and the addresses of certain registers. They should both be placed in the MSbyte. The
LSbyte is the data.
#define WHO_AM_I_ADDR_MSK 0x0F
#define CTRL_REG1_ADDR_MSK 0x20
#define CTRL_REG2_ADDR_MSK 0x21
#define CTRL_REG3_ADDR_MSK 0x22
#define HP_FILTER_RESET_MSK 0x23

#define OUTX_L_ADDR_MSK 0x28
#define OUTX_H_ADDR_MSK 0x29
#define OUTY_L_ADDR_MSK 0x2A
#define OUTY_H_ADDR_MSK 0x2B
#define OUTZ_L_ADDR_MSK 0x2C
#define OUTZ_H_ADDR_MSK 0x2D

#define READ_WO_AUTO_INCREMENT_MSK 0x80
#define READ_AND_AUTO_INCREMENT_MSK 0xC0U
#define WRITE_WO_AUTO_INCREMENT_MSK 0x00
#define WRITE_AND_AUTO_INCREMENT_MSK 0x40

#define DATA_RATE_40HZ_MSK 0x00 //(0000 0000)
#define DATA_RATE_160HZ_MSK 0x40 //(0100 0000)
#define DATA_RATE_640HZ_MSK 0x80 //(1000 0000)
#define DATA_RATE_2560HZ_MSK 0xC //(1100 0000)

enum data_rate_samples { HZ_40, HZ_160,      HZ_640,      HZ_2560 };
enum hpc {HPC_512, HPC_1024, HPC_2048, HPC_4096};
```

```
enum scale {G2,G6};
enum nbits {BITS_12, BITS_16};
enum das {BITS_12_RJ, BITS_16_LJ};
enum filter_mode {FILTER_ENABLED, FILTER_DISABLED};

typedef struct _accel_data
{
      uint8_t xh, xl, yh, yl, zh, zl;
}  accel_data;

//Buffer size for storing accelerometers data in a buffer instead of
//a structure
#define XYZ_BUF_SIZE 10 //6 bytes plus the end of packet

//Error codes
#define LIS_OK 0
#define LIS_ERROR_BUS_BUSY 1

#define pdFALSE    0
#define pdTRUE     1


//--------------Function declarations----------------------
//portCHAR lis_bus_select(void);
//void lis_bus_unselect(void);
uint8_t lis_perror(void);
portCHAR lis_init(void);
uint8_t lis_read_reg(uint8_t addr);
portCHAR lis_write_reg(uint8_t addr, uint8_t regval);

portCHAR lis_power_up(void);
portCHAR lis_power_down(void);
portCHAR lis_SetFilterDataSelection(enum filter_mode, enum hpc hpc_target);
portCHAR lis_SetScale(enum scale scale_trg);
portCHAR lis_SetDataAlignmentSelection(enum das das_trg);
portCHAR lis_SetDecFactor(enum data_rate_samples drs);

uint16_t lis_read_x(void);
uint16_t lis_read_y(void);
uint16_t lis_read_z(void);

uint16_t lis_read_xyz_b(uint8_t buf[XYZ_BUF_SIZE]);

#endif /*LIS3LV02DRIVER_H_*/
```

lis3l02_driver.c

```
/*
 * Accelerometer driver written by Antti Liesjärvi
 */

#include "lis3lv02_driver.h"
#include "i2c.h"
#include "rprintf.h"

/* Initializes the accelerometer
 *
 * In case of problems with the initialization the function sets the lis_errno
 * variable with the corresponding error code
 *
 * \return pdTRUE correct initialization
 * \return pdFALSE problems with the initialization
 */
portCHAR lis_init(void) {

	//Initialize control registers
	//CTRL_REG1(20h)
	//PD1=1
	//PD0=1		:device ON
	//DF1=0		:
	//DF0=0		:40Hz data rate
	//ST=0		:Self test OFF
	//Zen=1		:Enable axis Z
	//Yen=1		:Enable axis Y
	//Xen=1		:Enable axis X
	//

	//CTRL_REG2(21h)
	//FS=0		:Scale 2g
	//BDU=0		:Block Data Update enabled
	//BLE=0		:little endian mode
	//BOOT=0	:No reboot memory content
	//IEN=X		:Data ready on pad
	//DRDY=0	:Disable Data Ready Generation
	//SIM=0		:Serial mode interface to 4-wire SPI mode
	//DAS=1		:Data alignment 16 bit left justified

	//CTRL_REG3(22h)
	//ECK=0		:external clock disabled
	//HPDD=0	:High pass filter for direction detection disabled
	//HPFF=0	:High pass filter for free fall disabled
	//FDS=0		:Filtered data section bypassed
	//res
	//res
	//CFS1=0
	//CFS0=0	:High-pass filter Cut off frequency selection to 512


	lis_write_reg(CTRL_REG1_ADDR_MSK, 0xC7);
	lis_write_reg(CTRL_REG2_ADDR_MSK, 0x41);
```

```
	lis_write_reg(CTRL_REG3_ADDR_MSK, 0x00);
	//    lis_errno=LIS_OK;

	return (pdTRUE);
}

/* Reads a single register from the accelerometer
 *
 * \param address of the register to read
 * \return the value of the register read
 * \return 0xFF in case of error
 *
 *In case of error the function sets the lis_errno
 * variable with the corresponding error code
 */
uint8_t lis_read_reg(uint8_t addr) {

	unsigned char regval;

	i2c_send_byte(LIS_ADDR, READ_WO_AUTO_INCREMENT_MSK | addr);//(mode|addr)
	i2c_receive_byte(LIS_ADDR, &regval);

	return regval;
}

/* Writes a single register of the accelerometer
 *
 * \param address of the register to write
 * \param value to be written in the register
 *
 * \return pdTRUE if success
 * \return pdFALSE if error
 *
 * In case of error the function sets the lis_errno
 * variable with the corresponding error code
 */
portCHAR lis_write_reg(uint8_t addr, uint8_t regval) {

	unsigned char table[2];

	table[0] = addr | WRITE_WO_AUTO_INCREMENT_MSK;
	table[1] = regval;

	i2c_send(LIS_ADDR, 2, table);

	return (pdTRUE);
}

portCHAR lis_power_down(void) {
	uint8_t conf;

	//Read the actual configuration
	conf = lis_read_reg(CTRL_REG1_ADDR_MSK);
	if (conf == pdFALSE) {
		// debug("POWER DOWN could not get the bus\n");
```

```
			return (pdFALSE);
		} else {
			// Modify only the power down bits in the control register
			lis_write_reg(CTRL_REG1_ADDR_MSK, conf & 0x3F);//(0011 1111)
			return (pdTRUE);
		}
}

portCHAR lis_power_up(void) {
		uint8_t conf;

		conf = lis_read_reg(CTRL_REG1_ADDR_MSK);
		if (conf == pdFALSE) {
			// debug("POWER UP could not get the bus\n");
			return (pdFALSE);
		} else {
			// Modify only the power down bits in the control register
			lis_write_reg(CTRL_REG1_ADDR_MSK, conf | 0xC0); //(1100 0000)
			return (pdTRUE);
		}
}

portCHAR lis_SetDecFactor(enum data_rate_samples drs) //Data Rate Samples
{
		uint8_t conf, conf2 = 0;

		conf = lis_read_reg(CTRL_REG1_ADDR_MSK);
		if (conf == pdFALSE) {
			// debug("SET DEC FACTOR could not get the bus\n");
			return (pdFALSE);
		} else {
			//modify only the decimation factor bits in the control register
			switch (drs) {
			case HZ_40:
				conf2 = conf & 0xCF; //(XX00 XXXX)
				break;
			case HZ_160:
				conf2 = ((conf & 0xCF) | 0x10); //(XX01 XXXX)
				break;
			case HZ_640:
				conf2 = ((conf & 0xCF) | 0x20); //(XX10 XXXX)
				break;
			case HZ_2560:
				conf2 = conf | 0x30; //(XX11 XXXX)
				break;
			default:
				conf2 = conf;
			}
			lis_write_reg(CTRL_REG1_ADDR_MSK, conf2);

			return (pdTRUE);
		}
}

portCHAR lis_SetFilterDataSelection(enum filter_mode f_mode,
```

```
        enum hpc hpc_target) {
    uint8_t conf = 0, conf2 = 0;

    conf = lis_read_reg(CTRL_REG3_ADDR_MSK);
    if (conf == pdFALSE) {
        // debug("SET FILTER could not get the bus\n");
        return (pdFALSE);
    } else {
        switch (f_mode) {
        case FILTER_DISABLED:
            conf2 = conf & 0xEF; //(XXX0 XXXX)
            break;
        case FILTER_ENABLED:
            conf2 = conf | 0x10; //(XXX1 XXXX)
            break;
        default:
            conf2 = conf;
        }
        switch (hpc_target) {
        case HPC_512:
            conf2 = conf2 & 0xFC; //(XXXX XX00)
            break;
        case HPC_1024:
            conf2 = ((conf2 & 0xFC) | 0x01); //(XXXX XX01)
            break;
        case HPC_2048:
            conf2 = ((conf2 & 0xFC) | 0x02); //(XXXX XX10)
            break;
        case HPC_4096:
            conf2 = conf2 | 0x03; //(XXXX XX11)
            break;
        default:
            break;
        }
        lis_write_reg(CTRL_REG3_ADDR_MSK, conf2);
        lis_read_reg(HP_FILTER_RESET_MSK); //Reset the contents of the internal
                                           //filter
        return (pdTRUE);
    }
}

portCHAR lis_SetScale(enum scale scale_trg) {
    uint8_t conf, conf2;

    conf = lis_read_reg(CTRL_REG2_ADDR_MSK);
    if (conf == pdFALSE) {
        // debug("SET SCALE could not get the bus\n");
        return (pdFALSE);
    } else {
        // Modify only the decimation factor bits in the control register
        switch (scale_trg) {
        case G2:
            conf2 = conf & 0x7F; //(0XXX XXXX)
            break;
        case G6:
```

```
                conf2 = conf | 0x80; //(1XXX XXXX)
                break;
            default:
                conf2 = conf;
            }
            lis_write_reg(CTRL_REG2_ADDR_MSK, conf2);

            return (pdTRUE);
        }
}

portCHAR lis_SetDataAlignmentSelection(enum das das_trg) {
        uint8_t conf, conf2;

        conf = lis_read_reg(CTRL_REG2_ADDR_MSK);
        if (conf == pdFALSE) {
            // debug("SET SCALE could not get the bus\n");
            return (pdFALSE);
        } else {
            // Modify only the decimation factor bits in the control register
            switch (das_trg) {
            case BITS_12_RJ:
                conf2 = conf & 0xFE; //(XXXX XXX0)
                break;
            case BITS_16_LJ:
                conf2 = conf | 0x01; //(XXXX XXX1)
                break;
            default:
                conf2 = conf;
            }
            lis_write_reg(CTRL_REG2_ADDR_MSK, conf2);

            return (pdTRUE);
        }
}

uint16_t lis_read_x(void) {

        unsigned char recvdata[2];
        unsigned char senddata = READ_WO_AUTO_INCREMENT_MSK | OUTX_L_ADDR_MSK;
        //(mode|addr)

        i2c_send_byte(LIS_ADDR, senddata);
        i2c_receive(LIS_ADDR, 2, recvdata);

        //i2c_transfer(LIS_ADDR, 1, &senddata, 2, recvdata);

        return recvdata[0] | recvdata[1] << 4;
}

uint16_t lis_read_y(void) {
        unsigned char recvdata[2];
        unsigned char senddata = READ_WO_AUTO_INCREMENT_MSK | OUTY_L_ADDR_MSK;
        //(mode|addr)
```

```
	i2c_send_byte(LIS_ADDR, senddata);
	i2c_receive(LIS_ADDR, 2, recvdata);

	//i2c_transfer(LIS_ADDR, 1, &senddata, 2, recvdata);

	return recvdata[0] | recvdata[1] << 4;

}

uint16_t lis_read_z(void) {
	unsigned char recvdata[2];
	unsigned char senddata = READ_WO_AUTO_INCREMENT_MSK | OUTZ_L_ADDR_MSK;
	//(mode|addr)

	i2c_send_byte(LIS_ADDR, senddata);
	i2c_receive(LIS_ADDR, 2, recvdata);

	//i2c_transfer(LIS_ADDR, 1, &senddata, 2, recvdata);

	return recvdata[0] | recvdata[1] << 4;

}

//The same than lis_read_xyz but puts the results in a buffer ready to be sent
uint16_t lis_read_xyz_b(uint8_t buf[XYZ_BUF_SIZE]) {

	i2c_send_byte(LIS_ADDR, READ_WO_AUTO_INCREMENT_MSK | OUTX_L_ADDR_MSK);//(mode|addr)
	i2c_receive(LIS_ADDR, 6, buf);

	return (pdTRUE);
}
```

canproto.h

```
#ifndef _CANPROTO_H_
#define _CANPROTO_H_

#include "config.h"
#include "can_lib.h"
//#include "controller.h"
// Flags

#define CANBUFFER_SIZE    10
// Size of the data buffer where received CAN messages are stored

// Message numbers
#define MCASK 0 // Number for asking data from motion controller
#define MCWRITE 1

// CAN message IDs
#define ASKDATA_ID 0b00100100000
#define REPLYDATA_ID 0b00101100000

#define MCASK_ID (ASKDATA_ID+MCID)
#define MCUREPLY_ID (REPLYDATA_ID+MCUID)


// Operation codes
#define ASKOPCODE   0xB004
#define ASKLONGOPCODE     0xB005

#define CODE_RESET 0x0402       //reset
#define CODE_ENDINIT      0x0020       //endinit
#define CODE_AXISON       0x0102       //axison
#define CODE_CACC         0x24A2       //command acceleration
#define CODE_CSPD         0x24A0       //command speed
#define CODE_CPOS         0x249E       //command position
#define CODE_UPD          0x0108             //update immediately
#define CODE_STOP         0x0184             //stop 2 motion
#define CODE_WAIT         0x0804       //wait
#define CODE_FAULTRESET   0x1C04       //reset faults
#define CODE_MC           0x0F70       //motion is completed
#define CODE_SAP          0x8400       //sets actual position to the command
                                 // value
#define CODE_MODE_TYPE    0x5909 //command mode type
#define CODE_CPA          0x2000FFFF   //absolute
#define CODE_CPR          0x0000DFFF   //relative
#define CODE_TUM0         0x0000BFFF
#define CODE_TUM1         0x4000FFFF   //keep position and speed reference
#define CODE_SP1          0x8301BBC1   //speed mode 1
#define CODE_PP3          0x8701BFC1   //position mode 3
#define CODE_PP1          0X8501BDC1   //position mode 1

#define ACC_01      0x0000199A   //value of acceleration = 0.1
#define ACC_03                   0x00004CCD   //value of acceleration = 0.3
#define ACC_1                    0x00010000   //value of acceleration = 1
```

```
#define ASK_APOS          0x0228       //ask actual position
#define ASK_ASPD          0x022C       //ask actual speed
#define ASK_CPOS          0x029E             //ask command position
#define ASK_POSERR 0x022A       //ask position error
/**Memory addresses of variables*/

// Structure for received CAN message data
typedef struct _canmess
{
 uint8_t request;
 uint8_t data[8];
 uint8_t dlc;
} CANmessage;


// Function prototypes
int caninit(void);
int canrxconfig(void);

int CANSendMessage(uint16_t id, uint8_t* data, uint8_t len,uint8_t irq);
int CANconfigRX (st_cmd_t *cs, uint16_t id, can_cmd_t cmd, uint8_t *buffer, uint8_t len,
uint8_t irq);

//uint8_t get_can_message(uint8_t id);
//void handle_can_messages();

uint8_t drive_write_param(uint16_t address, int16_t value);
uint8_t drive_write_param32(uint16_t address, int32_t value);
uint8_t drive_write_paramf(uint16_t address, int32_t value);
uint8_t drive_send_command(uint16_t code);
int16_t drive_read_param(uint16_t address);
int32_t drive_read_param32(uint16_t address);
void drive_init();

#endif //_CANPROTO_H_
```

canproto.c

```
/* 22.7.2009, Johannes Aalto
 * Modified CAN prototype model for using Microcontroller as a CAN master
 * and a Slave motor controller(s)
 */
#include <avr/io.h>
#include <avr/interrupt.h>
#include <inttypes.h>
#include <util/delay.h>
#include "canproto.h"
#include "can_drv.h"
#include "can_lib.h"
#include "global.h"
#include "rprintf.h"
#include "statemachine.h"

static st_cmd_t ask_cmd;
static uint8_t ask_data[8];
//static st_cmd_t req[16]; //global pointers to received messages

/*
 * Initializes the CAN module
 * @param none
 * @return 0
 */
int caninit(void) {

	can_init(0); // Initialize CAN controller (parameter has no effect on fixed
		// baudrate)
	CANGIE = 0x3A; // Enable CAN interrupts (receive, transmit and errors)
	canrxconfig(); // Initialize CAN messages to be received

	return 0;
}

/*
 * Prepares the CAN message objects for reception
 * @param none
 * @return 0
 */
int canrxconfig(void) {

	/*
	 * Prepares a CAN message object for reception
	 * @param *cs  pointer to the structure for message object information
	 * @param id  standard (11-bit) id of the message to be received
	 * @param command  driver command that defines type of reception
	 * @param *pointer to the buffer where received data can be stored
	 * @param len number of bytes to be received
	 * @param irq interrupt on reception (1=yes)
	 * @return 0*/
	// int CANconfigRX (st_cmd_t *cs, uint16_t id, can_cmd_t command, uint8_t    //
*buffer, uint8_t len, uint8_t irq)
```

```
	CANconfigRX(&ask_cmd, MCUREPLY_ID, CMD_RX_DATA_MASKED, ask_data, 2, 0);
	return 0;
}

/*
 * Prepares a CAN message for sending and gives the command to CAN driver
 * Blocks until the transfer is completed.
 * Frees the used MOB.
 * @param id  standard (11-bit) id of the CAN message
 * @param data  pointer to the array of data to be send
 * @param len  number of bytes to be send (max. 8)
 * @return 0
 */

int CANSendMessage(uint16_t id, uint8_t* data, uint8_t len, uint8_t irq) {

	st_cmd_t cs;
	cs.cmd = CMD_TX_DATA;
	cs.id.std = id;
	cs.dlc = len;
	cs.pt_data = data;
	cs.ctrl.ide = 0;
	cs.ctrl.rtr = 0;

	can_cmd(&cs);
	if (cs.handle < 8) {
		sbi(CANIE2,(cs.handle)*irq); //enable interrupt for the MOb
	} else {
		sbi(CANIE1,((cs.handle)-8)*irq);
	}
	// Wait until the transfer is completed and release mob
	while (can_get_status(&cs) != CAN_STATUS_COMPLETED)
		;
	return 0;
}

/*
 * Prepares a CAN message object for reception
 * @param *cs  pointer to the structure for message object information
 * @param id  standard (11-bit) id of the message to be received
 * @param command  driver command that defines type of reception
 * @param *pointer to the buffer where received data can be stored
 * @param len  number of bytes to be received
 * @param irq  interrupt on reception (1=yes)
 * @return 0
 */
int CANconfigRX(st_cmd_t *cs, uint16_t id, can_cmd_t command, uint8_t *buffer,
		uint8_t len, uint8_t irq) {

	cs->cmd = command;
	cs->id.ext = 0;
	cs->id.std = id;
	cs->dlc = len;
	cs->pt_data = buffer;
	cs->ctrl.ide = 0;
```

```
	cs->ctrl.rtr = 0;

	while (can_cmd(cs) != CAN_CMD_ACCEPTED)
		;
	if (cs->handle < 8) {
		sbi(CANIE2,(cs->handle)*irq); //Enable interrupt for the MOb
	} else {
		sbi(CANIE1,((cs->handle)-8)*irq);
	}
	return 0;
}

/* Write one parameter to drive 1 into specified drive memory address.
 *
 *TODO: Doesn't handle CAN errors correctly.*/
uint8_t drive_write_param(uint16_t address, int16_t value) {
	uint8_t ret = 0;
	uint8_t message[4];

	//POSERR=0   121 2A 20 00 00    set position error = 0

	message[0] = (uint8_t) address;
	message[1] = (uint8_t) (address >> 8);
	message[2] = (uint8_t) value;
	message[3] = (uint8_t) (value >> 8);

	ret = CANSendMessage(MCASK_ID, message, 4, 0);
	return ret;
}

/**Write one parameter to drive 1 into specified drive memory address.
 *
 *TODO: Doesn't handle CAN errors correctly.*/
uint8_t drive_write_param32(uint16_t address, int32_t value) {
	uint8_t ret = 0;
	uint8_t message[6];

	//POSERR=0   121 2A 20 00 00    set position error = 0
	/*In each 2 byte field, LSB is sent first.
	 * In 4 byte field, the More significant 2 bytes are sent first.*/
	message[0] = (uint8_t) address;
	message[1] = (uint8_t) (address >> 8);
	message[2] = (int8_t) value;
	value >>= 8;
	message[3] = (value);
	value >>= 8;
	message[4] = (uint8_t) (value);
	value >>= 8;
	message[5] = (uint8_t) (value);
	/*
	 message[4]=(int8_t)value;
	 value>>=8;
	 message[5]=(value);
	 value>>=8;
	 message[2]=(uint8_t)(value);
```

```
        value>>=8;
        message[3]=(uint8_t)(value);*/
       ret = CANSendMessage(MCASK_ID, message, 6, 0);
       return ret;
}

// Fixed point byte order differs from 32bit integer
uint8_t drive_write_paramf(uint16_t address, int32_t value) {
       int32_t ret = (int16_t) value;
       ret <<= 16;
       value >>= 16;
       ret += value;
       drive_write_param32(address, ret);
       return 0;
}

/* Ask for parameters of drive 1
 * from specified drive memory address.
 * Returns the read value.
 * TODO: Doesn't handle CAN errors correctly.
 */
int16_t drive_read_param(uint16_t address) {
       int16_t ret;
       uint8_t message[6];

       //?POSERR          121 04 B0 20 00 2A 02          162 04 08 2A 02 FF FF
       ask position error
       message[0] = (uint8_t) ASKOPCODE;
       message[1] = (uint8_t) (ASKOPCODE >> 8);
       message[2] = (uint8_t) (MCUID << 4);
       message[3] = (uint8_t) 0x00;
       message[4] = (uint8_t) address;
       message[5] = (uint8_t) (address >> 8);

       ret = CANSendMessage(MCASK_ID, message, 6, 0);

       // Wait until the transfer is completed and release mob
       while (1) {
              uint8_t status = can_get_status(&ask_cmd);

              if (status == CAN_STATUS_COMPLETED) {
                     while (can_cmd(&ask_cmd) != CAN_CMD_ACCEPTED)
                            ;

                     break;
              } else if (status == CAN_STATUS_ERROR) {
                     CANSTMOB = CANSTMOB & 0x0F;//clear mob errors
                     while (can_cmd(&ask_cmd) != CAN_CMD_ACCEPTED)
                            ;

                     break;
              } else {
              }
       }
```

```
	ret = ask_data[5];
	ret <<= 8;
	ret += ask_data[4];

	//	rprintfu16(ret);

	return ret;
}
/* Ask for parameters of drive 1
 * from specified drive memory address.
 * Returns the read value.
 * TODO: Doesn't handle CAN errors correctly.
 */
int32_t drive_read_param32(uint16_t address) {
	int32_t ret;
	uint8_t message[6];

	//?POSERR		121 04 B0 20 00 2A 02		162 04 08 2A 02 FF FF
	ask position error
	message[0] = (uint8_t) ASKLONGOPCODE;
	message[1] = (uint8_t) (ASKLONGOPCODE >> 8);
	message[2] = (uint8_t) (MCUID << 4);
	message[3] = (uint8_t) 0x00;
	message[4] = (uint8_t) address;
	message[5] = (uint8_t) (address >> 8);

	ret = CANSendMessage(MCASK_ID, message, 6, 0);

	// Wait until the transfer is completed and release mob
	while (1) {
		uint8_t status = can_get_status(&ask_cmd);

		if (status == CAN_STATUS_COMPLETED) {
			while (can_cmd(&ask_cmd) != CAN_CMD_ACCEPTED)
				;

			break;
		} else if (status == CAN_STATUS_ERROR) {
			CANSTMOB = CANSTMOB & 0x0F;//clear mob errors
			while (can_cmd(&ask_cmd) != CAN_CMD_ACCEPTED)
				;

			break;
		} else {
		}
	}

	ret = (uint8_t) ask_data[7];
	ret <<= 8;
	ret += (uint8_t) ask_data[6];
	ret <<= 8;
	ret += (uint8_t) ask_data[5];
	ret <<= 8;
	ret += (uint8_t) ask_data[4];
```

```
	//	rprintfu16(ret);

	return ret;
}

uint8_t drive_send_command(uint16_t code) {
	uint8_t ret = 0;
	uint8_t message[2];

	//POSERR=0   121 2A 20 00 00    set position error = 0

	message[0] = (uint8_t) code;
	message[1] = (uint8_t) (code >> 8);

	ret = CANSendMessage(MCASK_ID, message, 2, 0);
	return ret;
}

void drive_init() {

	drive_send_command(CODE_RESET);
	_delay_ms(500);
	drive_send_command(CODE_ENDINIT);
	_delay_ms(500);
	drive_send_command(CODE_AXISON);

}
```

config.h

```
#ifndef _CONFIG_H_
#define _CONFIG_H_


#define CAN_BAUDRATE            250    // CAN baudrate (baud)
                  // Used in CAN initialization macros to calculate baudrate
#define FOSC                     16000


// Motor Controller ID
#define MCID 1
// Microcontroller ID
#define MCUID 2

#endif
```

util.h

```
#ifndef UTIL_H
#define UTIL_H

#define cbi(A,B) A&= ~(1<<B)    // Clear bit B in byte A
#define sbi(A,B) A|= (1<<B)     // Set bit B in byte A

#endif
```